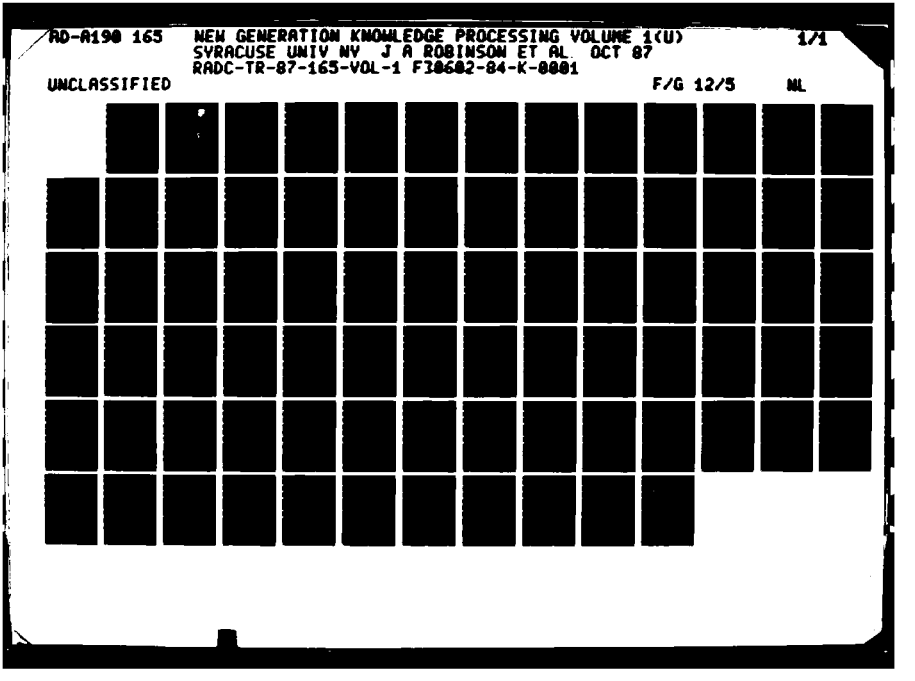
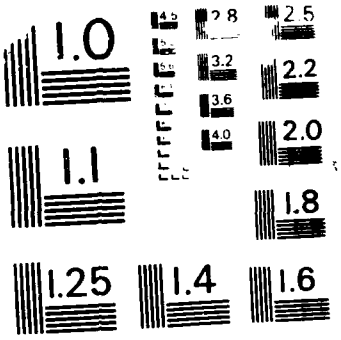

AD-A190 165 NEW GENERATION KNOWLEDGE PROCESSING VOLUME 1(U) 1/1
SYRACUSE UNIV NY J A ROBINSON ET AL OCT 87
RADC-TR-87-165-VOL-1 F30602-84-K-0001
UNCLASSIFIED F/G 12/5 NL



(4)

RADC-TR-87-165, Vol I (of three)
Final Technical Report
October 1987

NEW GENERATION KNOWLEDGE PROCESSING

Syracuse University

J. Alan Robinson and Kevin J. Greene

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-165, Vol I (of three) has been reviewed and is approved for publication.

APPROVED: Northrup Fowler III

NORTHRUP FOWLER III
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Richard W. Poulliot

RICHARD W. POULIOT
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS N/A
2a SECURITY CLASSIFICATION AUTHORITY N/A	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5 MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-165, Vol I (of three)
6a NAME OF PERFORMING ORGANIZATION Syracuse University 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c ADDRESS (City, State, and ZIP Code) Syracuse NY 13244	7b ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b OFFICE SYMBOL (If applicable) COES	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-84-C-0031
8c ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 62702F PROJECT NO: 5581 TASK NO: 27 WORK UNIT ACCESSION NO:
11 TITLE (Include Security Classification) NEW GENERATION KNOWLEDGE PROCESSING	
12 PERSONAL AUTHOR(S) J. Alan Robinson, Kevin J. Greene	
13a TYPE OF REPORT Final 13b TIME COVERED FROM Dec 83 TO Jan 87 14 DATE OF REPORT (Year, Month, Day) October 1987 15 PAGE COUNT 80	
16 SUPPLEMENTARY NOTATION N/A	

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence; Graph Reduction; Logic Programming; Combinators; Functional Programming; Programming Languages

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The main goal of this project was to design a high-level programming system (which we have named SUPER, an acronym for "Syracuse University Parallel Expression Reducer") with two parts: a language which would combine the functional (as in LISP, SASL or ML) with the relational (as in PROLOG) programming concepts into a single new paradigm and a machine which would execute programs written in the language, using reduction and a multiprocessor architecture.

The SUPER language is an extension of the basic lambda-calculus which we call lambda plus. It is formally a collection of expressions together with some rules and definitions which give them meaning and make it possible to do deductive reasoning and computation with them. The expressions of the SUPER language fall into three main syntactic categories: atoms, abstractions, and combinations.

(cont'd on reverse)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Northrup Fowler III	22b TELEPHONE (Include Area Code) (315) 330-7794 22c OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 Previous editions are obsolete

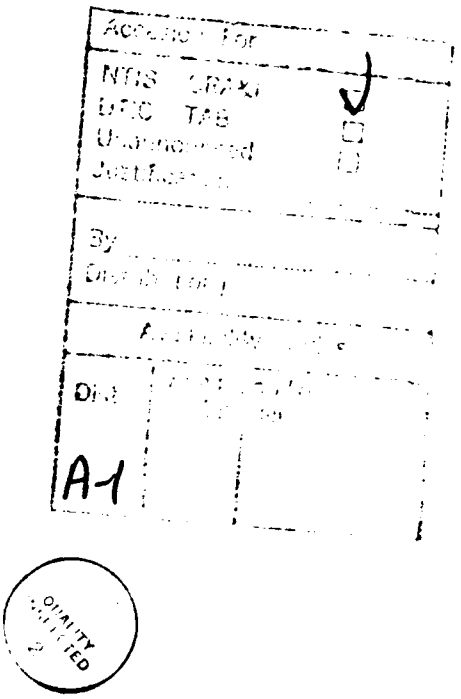
SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block 19. Abstract (Cont'd)

Volume I describes the SUPER system, and discusses the conceptual background in terms of which it can best be understood. In developing these ideas over the period of the project we devised and implemented two related single-processor reduction systems, LNF and LNF-Plus, as experimental tools to help us learn more about SUPER language design issues. These systems have turned out to be of considerable interest and utility in their own right, and they have taken on separate and independent identities.

Volume 2 contains a detailed presentation of the single-processor software programming system LNF which was developed to serve as a test bed and simulation tool for the "classical" part of the SUPER system.

Volume 3 presents the final, enhanced version of LNF, which we call LNF-Plus and which provides the user with as close an approximation as we can achieve on a single processor of the SUPER system. Volume 3 is also designed as a useful guide to someone who wishes to use the system for experimental computations.



CONTENTS OF VOLUME 1

CHAPTER 0	INTRODUCTORY REMARKS	1
CHAPTER 1	THE SUPER LANGUAGE	11
CHAPTER 2	GRAPH REDUCTION	30
CHAPTER 3	THE SUPER MACHINE	43
CHAPTER 4	RELATED WORK	69
	REFERENCES	70

CHAPTER 0. INTRODUCTORY REMARKS.

This is Volume 1 of a three-volume final report on a project entitled "New Generation Knowledge Processing" which began on 6 December 1983 and ended on 31 December 1986.

Goal of the project. The goal of the project was to design and develop a high-level programming system (called SUPER, for **S**yracuse **U**niversity **P**arallel **E**xpression **R**eduction) consisting of two parts:

- a language with both functional (as in LISP, SASL or ML) and relational (as in PROLOG) constructs;
- a machine to process expressions written in the language, using reduction and a multiprocessor architecture.

SUPER and LNF-Plus. As an experimental tool and test-bed to help us study SUPER design issues we developed a single-processor reduction system, LNF-Plus, which turned out to be of considerable interest and utility in its own right. LNF-Plus is a single-processor reduction-based implementation of the SUPER language. In order to emphasize and preserve its independence LNF-Plus is documented separately in the second and third volumes of this report. **Volume 2** deals with LNF, the purely functional programming subset of LNF-Plus. **Volume 3** discusses the relational programming features which extend LNF to LNF-Plus, and provides the necessary "how-to" instructions which enable a somewhat experienced Lisp Machine user to begin interacting with LNF-Plus.

Volume 1 contains, following this introductory **Chapter 0**, three main chapters. **Chapter 1** discusses the SUPER language. **Chapter 2** explains graph reduction. **Chapter 3** presents our ideas for the SUPER machine. These call for a fine-grained parallel machine architecture consisting of a large number (millions) of identical small processors ("nodules") linked by a dynamically variable connection network (the "postal system") enabling the nodules to act as the nodes in a graph representation of

a SUPER expression. The collective behavior of the nodules effects the reduction of the expression to normal form.

SUPER and LOGLISP. The immediate precursor of SUPER was LOGLISP (Robinson and Sibert [24]). The goal of LOGLISP was to design and experimentally implement a programming system in which both relational and functional styles were supported. To some extent this was achieved, but in a rather awkward way. The awkwardness was traceable to the fact that the relational programming part of LOGLISP was no more than a kind of patch grafted onto an existing LISP, so that LOGLISP is simply an old language with some modifications. SUPER, on the other hand, is a new language designed from scratch to meet the same goal. It is a natural successor to LOGLISP. Indeed, it is a second attempt to reach the same objectives. Thus programming in SUPER, as in LOGLISP, is meant to be a generalized kind of logic programming which subsumes both relational programming and functional programming.

We do not attempt to persuade the reader of the advantages of the relational programming style based on the Horn clause resolution subset of the first order predicate calculus. This has been well done both by its many proponents (especially by Kowalski [20] and Clark [7]) and by the practical success of PROLOG.

Nor do we try to make the case for the virtues of the functional programming style based on the λ-calculus. This has been already achieved not only by its many advocates, notably Landin [21], Burge [4], Turner [28], and Henderson [15], but also by the great practical success of LISP. Of course LISP is not a purely declarative λ-calculus system, since it contains several imperative control features. Nevertheless LISP has pioneered the trail for later, purer λ-calculus-based programming systems (including certain versions of LISP itself, such as SCHEME [26]) and has introduced several generations of programmers to the power and elegance of the functional programming style in which functions are dealt with as first-class objects on a par with

numbers, truth values, strings, and the like.

Our main purpose is rather to justify our view (as was attempted earlier with LOGLISP) that relational and functional programming need not and should not be kept apart in separate systems but can be smoothly integrated within one homogeneous linguistic framework. The following discussion gives some motivation for this view.

What does PROLOG do? There is a nice intuitive version of what PROLOG is actually doing, as it looks for answers, in a world described by a given sentence P (its "program"), to the query

$$Q \equiv \exists x C(x).$$

According to this version PROLOG is attempting to find a counterexample to Q, namely, a way of interpreting the language (relation and function symbols) found in P and Q such that P and (**not** Q) both come out true.

Note that **not** Q is the proposition

$$\forall x \textbf{ not } C(x).$$

Now in the event that Q is a logical consequence of P, the attempt to find a counterexample to Q will automatically generate one or more barriers

$$A_i \textbf{ because } R_i$$

in which A_i is an answer and R_i is a proof of $C(A_i)$ from P. There may even be infinitely many such barriers. Any one of them is enough to bar the proposition **not** Q, that is, to show that Q logically follows from P.

Proofs = computation traces. PROLOG users think of the various barriers individually, as the different results of many (small) computations caused by the submission of the query Q in the context of P. However, only the answer A_i in each barrier is actually of immediate interest, while R_i is simply the justification of that answer. Thus a PROLOG transaction has the overall form

user	P
machine	**ok**
user	**is there an** x **such that** C(x)?
machine	**yes**: $x = A_1$
user	**why?**
machine	**because** R_1
user	**any others**?
machine	**yes**: $x = A_2$
user	**any others**?
:	: : : :

where the sequence of requests for further answers may eventually end with the exchange

user	**any others**?
machine	**no**

or may continue indefinitely (because there are infinitely many answers). The machine may even silently pursue its search for "one more" answer for ever in vain, if there are no more answers but no way of knowing that this is so.

How does that work in LOGLISP? LOGLISP users, on the other hand, think of the set of barriers

$\{ A_i$ **because** $R_i \mid i = 1,\ldots, \}$

collectively as the (only) result of the single (large) computation caused by the same submission. Thus a LOGLISP transaction typically has the following overall form

user	P
machine	**ok**
user	**what is** $\{ x \mid C(x) \}$ **?**
machine	**it is** $\{ A_1,\ldots, A_n \}$

user **why A_i ?**

machine **because R_i**

when there are only finitely many answers (and in the special case that there are no answers the empty set { } would be returned). If there are infinitely many answers, a LOGLISP interpreter cannot (of course!) finish computing out the explicitly listed set of them all, and instead might be arranged to respond

machine **it is** $\{ A_1,..., A_n \} \cup B$

where B is a "remainder" expression describing the (infinite) set of remaining answers.

Proofs and traces normally needed only to remove doubts or locate errors. The proof (or justification) component R in a barrier (A **because** R) is the analog, for a logical computation, of the trace of an ordinary computation. As such, it is not normally wanted as part of the output, but is held in reserve in case the answer component A should be thought to need justification. This could be very important, for example, in the case of an expert system whose answers are either costly, or risky, or in some other way call for a supporting rationale. As we shall see, in the SUPER system as we presently envisage it, the proof components are not preserved and hence are not available to the user. This is because the SUPER process of reduction is "inhumanly" parallel in nature and its mere history would in general be of little use as an epistemological crutch.

Point of computation is to get output or to achieve effect. In general, the purpose of a computation is to obtain an output (or to achieve some effect, an output in a broader but still legitimate sense). Our reason for doing the computation is not normally to get its trace. Similarly, in logic, the main reason for doing a proof is not so much to have the proof itself as to be properly convinced of the truth of its conclusion.

Constructive proofs as natural source of outputs. A proof may or may not be constructive in character. A constructive proof is one which a construction is supplied

tor any object which is asserted to exist in the course of the proof. A nonconstructive proof that a certain kind of object exists is an argument which manages to convince its user of this by indirect means short of actually constructing such an object. Whatever their desirability in mathematics may be thought to be, constructive proofs are absolutely indispensable in logical computing. But we value the constructive proof less because it is a proof and more because it provides, as a side effect of its main task, an answer to the query we submitted.

Logic programming. Although the phrase "logic programming" has come to be associated mainly with PROLOG and its underlying Horn clause resolution logic, the concept is a more general one and can be discerned, for example, in LISP, SASL, and KRC. In those systems, however, the underlying logic is different. It is the equality logic of "substituting equals for equals to get equals" combined with that of abstracting and instantiating syntactic patterns using Church's lambda notation.

Definitions, axioms. The assertions in a logic program are often thought of as definitions, or axioms. But we can also think of a functional program in the same way. For example, the equation

factorial = (**Y** λf λn **if** (**zerop** n) **then 1 else** (**times** n (f (**sub1** n))))

is thought of as defining the factorial function, as indeed it does in the sense that, with this equation and the equations

Y F = F (**Y** F);

if true then A **else** B = A; **if false then** A **else** B = B;

zerop 0 = **true**; **zerop 1** = **false**; **zerop 2** = **false**; etc.,

times = λmλn **if** (**zerop** m) **then 0 else** (**plus** n (**times** (**sub1** m) n));

plus = λmλn **if** (**zerop** m) **then** n **else** (**add1** (**plus** (**sub1** m) n))

(**add1 0**) = **1**; (**add1 1**) = **2**; (**add1 2**) = **3**; etc.,

(**sub1 1**) = **0**; (**sub1 2**) = **1**; (**sub1 3**) = **2**; etc.,

as axioms, the beta- and delta-contraction rules allow us to make deductions like

(**factorial 13**) = ... = **6227020800**.

Of course, in practice we cannot actually store infinitely many defining equations for **add1**, **sub1**, **zerop**, and the like, and instead we simulate this by suitable algorithms based on the positional notation for the numerals, and we normally treat the functions **plus** and **times** in a similar fashion.

Function calls = theorems to be proved. Within this general logical framework, then, it is possible to explain (and indeed, to execute) function calls - like the call

(factorial 13)

above - by making logical deductions whose steps are constructive inferences based on the assertions (alias definitions, alias axioms) explicitly or implicitly included in the program. We are so accustomed to this mode of reasoning (or computation) that we do not think of it as a species of logical deduction, or theorem proving. But it is. The theorem proved above is the equation

(**factorial 13**) = 6227020800

and the proof is the entire trace of the computation of the output **6227020800** from the input (**factorial 13**). We can also think of the entire computation as a constructive proof of the general theorem (query)

there is an n **such that** (**factorial 13**) = n

and the computation simply a response to a user's request to prove this proposition. The proof is made, and, being constructive, carries along with it the supporting construction that n = **6227020800**, which provides the output. The proof itself is discarded or ignored. In fact in this example, which is of course theoretical and pedagogical, the pure proof is an enormously large thing because of the pure treatment of arithmetical functions and predicates. If we build these in and algorithmically exploit the positional notation for the numerals in the usual way, then intermediate steps involving such function calls as the expression (**times 7 720**) would be immediate, one-step transactions as if the equation

(**times 7 720**) = 5040

were one of the exioms.

Outputs = supporting constructions. The idea that in such logical reconstructions of computations the outputs are obtained from the supporting constructions of proofs of propositions in existential form is a basic one in our view of general logic programming in all its varieties. Functional programming, as in LISP and all its lambda-calculus based cousins (SASL, KRC, ML, even APL, et al.), is no different in this respect from relational programming, as in PROLOG, where this aspect of the logical reconstruction is much more obvious (being hardly disguised at all). PROLOG's origins in mechanical theorem proving are very much worn on its sleeve. In our efforts to develop a uniform concept of logic programming which comfortably fits both the LISP-like and PROLOG-like paradigms, we have found that it is the "denotation preserving" scheme of equational deduction, or reduction, for short, which serves best to capture both. If we regard a PROLOG-like computation as a succession of transformations applied to a series of set expressions, without changing the set denoted by them, we have the same basic paradigm as in the LISP-like case. So we think of PROLOG-like computations as

$$\{ x \mid C(x) \} = \ldots = \{ A_1, \ldots, A_n \} \cup B$$

with the "remainder" B often being the expression $\{\}$ (in case of finite set outputs) but in general being an expression capable of more development (possibly infinitely much more) by further transformations. This change of viewpoint can be made smoother by following the suggestion of Keith Clark that the set of n definite clauses in one's program which collectively define a given relation symbol R - in the sense that their conclusions all have the form $R(t_1 \ldots t_k)$ - can be reorganized into a single equational definition of R

$$R = \lambda (x_1 \ldots x_k) (G_1 \textbf{ or} \ldots \textbf{or } G_n)$$

that is to say (equivalently)

$$\forall (x_1 \ldots x_k) [R(x_1 \ldots x_k) \textbf{ iff } (G_1 \textbf{ or} \ldots \textbf{or } G_n)] \qquad (*)$$

where each G_i expresses the content of the ith clause as follows. If the ith clause is

$$\forall (y_1 \ldots y_m) \textbf{ if } (M_1 \textbf{ and} \ldots \textbf{and } M_q) \textbf{ then } R(t_1 \ldots t_k)$$

then G_i is

$$\exists (y_1 \ldots y_m) (x_1 = t_1 \text{ and } \ldots \text{and } x_k = t_k \text{ and } M_1 \text{ and } \ldots \text{and } M_q). \qquad (**)$$

Clark's idea is called "completion of the knowledge base". This is because, strictly speaking, the conjunction of the n clauses is logically equivalent not to (*) above, but only to the weaker assertion

$$\forall (x_1 \ldots x_k) [R(x_1 \ldots x_k) \text{ if } (G_1 \text{ or} \ldots \text{or } G_n)] \qquad (***)$$

with merely the **if**, rather than the stronger **iff**. The stronger assertion (*) provides a means of inferring negated R-sentences, since (*) is logically equivalent to the assertion

$$\forall (x_1 \ldots x_k) [\text{not} R(x_1 \ldots x_k) \text{ iff } (\text{not} G_1 \text{ and } \ldots \text{and not} G_n)]$$

where each component $\text{not}G_i$ of whose right hand side is (negating (**) above)

$$\forall (y_1 \ldots y_m) \text{ if } (x_1 = t_1 \text{ and } \ldots \text{and } x_k = t_k) \text{ then not } (M_1 \text{ and } \ldots \text{and } M_q).$$

This shows that in order to prove $\text{not}R(x_1 \ldots x_k)$ it is necessary to prove n distinct universally quantified theorems. This is why the "negation as failure" principle in PROLOG is so full of subtleties: the proof method used in PROLOG is in general not designed to prove such theorems.

For the purposes of our present discussion we do not need to pursue this point further. We are interested only in the idea of defining a relation R by means of an equation

$$R = \lambda(x_1 \ldots x_k)[G_1 \text{ or} \ldots \text{or } G_n]$$

just as we define a function F by means of an equation

$$F = \lambda(x_1 \ldots x_k) B.$$

It is this idea which makes it possible for us to combine relational with functional programming in the SUPER language and reduction system.

SASL, KRC based on reduction; LISP should be but is not. David Turner's well known functional programming systems SASL and KRC are reduction systems. This means

that they are logic programming systems based on the idea of proofs as equational deductions. The user types in an expression A as input, and after some work by the machine receives an expression B as output. The machine has reduced A to B, and thereby (as a side effect, so to speak) proved the equation A = B. The idea of reduction is of great importance but is really a very simple one - far simpler than the elaborate alternative idea based on denotational semantics and evaluation.

CHAPTER 1. THE SUPER LANGUAGE.

In this chapter we review the general logical background and underlying principles of the SUPER language.

The SUPER language: expressions and contractions. The SUPER language contains both the predicate calculus and the λ-calculus. Formally, it is a collection of expressions, together with some (informal) conventions which give them meaning and some (formal) inference rules, called contractions, which make it possible to do deductive reasoning and computation with the expressions.

A SUPER expression is either a symbol, or an abstraction, or a combination. Every expression has an arity, which is a nonnegative integer.

Symbols. Symbols are words of one or more characters; for example

not, and, or, some, true, false,
S, K, I, Y, 34, -23.6, plus,
x, y, zorro, a, banana, P, QUERY,

are symbols. It is part of the definition of each symbol that it is either a function symbol or a relation symbol, that it is either constant or variable, and that it has a given arity. Constants are written in **bold style**. Variables are written in plain style.

Abstractions. An abstraction has a bound variable, which is a variable, and a body, B, which is an expression. Its arity is one larger than that of B.

Combinations. A combination has an operator F of positive arity and an operand A, both of which are expressions. Its arity is one less than that of F. If F is an abstraction with bound variable V then the arities of V and A must be the same.

Free and bound occurrences of variables. An occurrence of a variable V in an expression E is a bound occurrence of V in E if and only if it is in a subexpression of E which is an abstraction whose bound variable is V (and otherwise it is a free occurrence of V in E).

Abstract vs. concrete syntax. The above syntax is abstract: it does not specify any particular concrete representation for expressions, and is in fact compatible with a wide variety of particular ways of writing or representing them. We will be concerned with two main ways of representing expressions: as strings and as graphs.

String representation. The string representation of an expression E is defined as follows:

- if E is a symbol then the string representation of E is E itself;
- if E is an abstraction with bound variable V and body B, then the string representation of E consists of the lower case greek letter λ, followed by the symbol V, followed by the string representation of B;
- if E is a combination with operator F and operand A, then the string representation of E consists of the lower case greek letter α, followed by the string representation of F, followed by the string representation of A.

There is in addition a wide variety of "sugared" (Landin [21]) ways of writing and displaying expressions, all representing the same underlying abstract syntax. For example the "beta-redex", whose string representation is

αλxB A

can also be (and most often is) written

(λxB)A

without the α, using juxtaposition, supplemented by parentheses if necessary, to

indicate combinations; but it can also be written in the **let** notation

let x **be** A **in** B

and in the **where** notation

B **where** x = A.

Thus $\alpha F \alpha G X$ can be written as F(GX), and $\alpha\alpha FGX$ as (FG)X or even as FGX (following a convention of "association to the left"). Since iterated combinations (*combinations whose operators are combinations*, etc.) are common we shall often write α^n as an abbreviation for a succession of n α's. **Chapter 3** of **Volume 2** discusses the many such "sweetened" syntactic variations which are recognized by LNF-Plus.

Graph representation. Inside the SUPER machine, and in the LNF-Plus interpreter, expressions are represented by rooted directed graphs, similar to LISP S-expressions, which may have both confluences ("sharing") and cycles, and may even contain redundant nodes inaccessible from the root ("garbage"). These graphs are discussed in more detail in **Chapter 2** and **Chapter 3** of the present volume, and throughout **Volume 2**.

Logical constants. Certain constant symbols are called logical constants and have a fixed role in the language. The following logical constants are relation symbols:

symbol	arity
true	0
false	0
not	1
and	2
or	2
equals	2
some	2

Other constants. There are the constant relation symbols:

symbol	arity
less	2
greater	2
pairp	1

the constant function symbols:

symbol	arity
if	3
numerals	0
sum	2
difference	2
product	2
quotient	2
head	1
tail	1

and the list-structure function symbols ("constructors"):

symbol	arity
pair	2
nil	0

Expression structure: heads and arguments. The string notation is especially useful for revealing certain structural features of expressions. Every expression E has a structure which in the string representation has the form

$$E = \alpha^m H A_1 \ldots A_m$$

for some $m \geq 0$, and expressions H, $A_1, \ldots, A_m$ such that H is not a combination and has arity $\geq m$. H is called the head of E, and A_i is called the i^{th} argument of E. Clearly, if m = arity of H then the arity of E is 0. Since H is not a combination it must be either a symbol or an abstraction.

Predicates. Sentences. A symbol is a predicate if and only if it is a relation symbol. An abstraction is a predicate if and only if its body is a predicate. Predicates of arity 0 are sentences. The general idea is that if P is a predicate of arity n then all 0-ary expressions of the form

$$\alpha^n P A_1 \ldots A_n$$

are sentences, and conversely that every sentence is a 0-ary expression of that form. Notice that in particular the logical constants **true** and **false** are sentences. They are also called truth values.

Not all sentences must make sense semantically, as is also the case in natural languages. The arities do not make up a full typing system, and so the arguments and operators in an expression might be meaningful in themselves but not appropriate in combination. For example, the expression $\alpha\alpha$ **plus true 3** is a well-formed expression as far as arities are concerned but lacks semantic significance (unless we extend the definition of **plus** beyond its customary domain).

The combination α^2**some** n P is a sentence. It makes sense only if P is an n-ary predicate, in which case it expresses the proposition that P applies to at least one n-tuple of things. Intuitively, α^2**some** n P expresses the same proposition as the more familiar formula $\exists V_1 \ldots V_n (P\ V_1 \ldots V_n)$, where $V_1 \ldots V_n$ are any n distinct variables not occurring free in P.

Tuple notation for abstractions and set notation for predicates. An abstraction whose string representation is

$$\lambda z_1 \ldots \lambda z_k B$$

where the z_i are distinct variables, may also be written in the "tuple" notation

$$\lambda(z_1 \ldots z_k) B$$

and if B is a sentence, the abstraction (which is therefore a predicate) may also be written in the "set" notations

$\{ (z_1 \ldots z_k) \mid B \}$, **setof** $(z_1 \ldots z_k)$ B.

Sets and predicates considered the same. The SUPER language does not distinguish between predicates and sets. It treats these notions as alternative but equivalent versions of the same notion. However, for reasons which will become clearer in the sequel, we will make "official" use of the alternative **setof** notation under certain circumstances.

Conjunctions, disjunctions, negations. The sentence **true** is (also known as) the empty conjunction. A nonempty conjunction is a sentence of the form

α^2**and**$A_1 \ldots \alpha^2$**and**A_n**true**

for some $n \geq 1$, where the A_i are sentences, the conjuncts of the conjunction. If $n = 1$ the conjunction may be identified with its (only) conjunct.

The sentence **false** is (also known as) the empty disjunction. A nonempty disjunction is a sentence of the form

α^2**or**$A_1 \ldots \alpha^2$**or**A_n**false**

for some $n \geq 1$, where the A_i are sentences, the disjuncts of the disjunction. If $n = 1$ the disjunction may be identified with its (only) disjunct.

A negation is a sentence of the form

α**not**A

where A is a sentence.

Existential quantifications. Sentences of the form α^2**some** n $\lambda x_1 \ldots \lambda x_n$B may be written

$\exists x_1 \ldots x_n$B.

Atomic sentences, equations, terms. Consider a 0-ary expression E of the form

$$\alpha^n H A_1 \ldots A_n$$

for some $n \geq 0$, (where the head H of the expression is therefore n-ary). When E is a sentence we shall be particularly concerned with the case when the H is an abstraction, or a symbol other than **and**, **or**, **not**, or **some** (that is to say, with sentences other than conjunctions, disjunctions, negations and quantifications). E may be written as the "term"

$$H(A_1, \ldots, A_n).$$

When its H is not an abstraction (and is therefore a symbol) E is called an atomic expression, and if E is a sentence, an atomic sentence. When in particular n = 2 and H is the constant **equals**, E is also called an equation.

An atomic expression $\alpha^2 HAB$ whose head is 2-ary may also be written in the infix notation

A H B

(writing the head between the two arguments) and in particular an equation may be written

A = B.

λ-normal form. A fundamentally important notion in the theory and applications of *SUPER* is that of *λ-normal form. An expression is said to be* in λ-normal form, or to be a λ-normal expression, if it contains no subexpressions of either of the following forms:

- $\alpha\lambda VBA$ (a "beta-redex")
- $\lambda V\alpha MV$ (an "eta-redex")

where M is an expression not containing free occurrences of the variable V. An expression which is not in λ-normal form can often be transformed into an equivalent λ-normal one by persistently applying the operation of λ-contraction (see immediately below) until no redexes of either kind remain.

λ-contractions and λ-computations. For every beta-redex

$$\alpha\lambda VBA$$

there is a corresponding "beta-contractum", namely the expression obtained by substituting the expression A for each free occurrence of the variable V in B. For every eta-redex

$$\lambda V\alpha MV$$

there is a corresponding "eta-contractum", namely the expression M. A λ-contraction operation can be performed on any expression E which is not in λ-normal form, by identifying a set of occurrences of beta- or eta-redexes in E and replacing each of them with an occurrence of its corresponding contractum. Since in general a non-λ-normal expression may have n such redexes, it may be λ-contracted in 2^n ways.

A λ-computation starting with E is then a (possibly infinite) sequence

$$E_1, E_2, \ldots,$$

of expressions in which E_1 is E, and each expression after the first is obtained by λ-contracting its predecessor. The λ-computation is complete if it is finite and its last expression is λ-normal. All complete λ-computations starting with an expression E end with essentially the same expression, which is called the λ-normal form of E. This means, roughly, that we obtain the λ-normal form of E by choosing successive λ-contractions in any way we like, until no further λ-contractions are possible (the Church-Rosser property). More exactly, we have to make the choices so as to avoid nonterminating computations. This can always be done, provided that a complete computation exists, for example by always contracting the leftmost redex in E_i to get E_{i+1}. λ-computation is an example of computation viewed as the reduction of expressions to normal form.

Normal forms in general. A λ-reduction machine is a machine which accepts an expression as input and returns its λ-normal form as output. Presumably, the machine's design is based on some particular algorithm for systematically constructing a complete λ-computation starting with a given expression as input (say, the leftmost redex algorithm mentioned above). In this report we discuss a somewhat more complicated reduction machine, which carries out SUPER-computations rather than λ-computations. SUPER-computations involve 49 kinds of redex contraction, which are displayed in Figure 1. One of them (number 1) is eta-contraction, but beta-contraction is not among them. In its place are contractions (1 through 20) which collectively have the same effect.

	REDEX	CONTRACTUM	REMARKS
1	$\lambda x(F\ x)$	F	x not free in F
2	$\lambda x(F\ x)$	W λxF	x free in F
3	$\lambda x x$	I	
4	λxC	KC	x not free in C
5	$\lambda x(FA)$	S λxF λxA	x free in both F and A
6	$\lambda x(FA)$	C λxF A	x free in F but not in A
7	$\lambda x(FA)$	B F λxA	x free in A but not in F
8	C F X Y	F Y X	F not of the form (B C D)
9	C (B F G) X	C F G X	
10	C W F X Y	W (F Y) X	
11	B F G X	F (G X)	G not of the form (B C D)
12	B F (B G X)	B F G X	
13	B W F G X	W (F (G X))	
14	S W F G X	W (F X) (G X)	
15	S (B F G) X	S F G X	
16	S F G X	(F X) (G X)	F not of the form (B C D))
17	K X Y	X	
18	I X	X	
19	W F X	F X X	
20	Y F	F(Y F)	
21	head (pair A B)	A	
22	tail (pair A B)	B	
23	pairp (pair A B)	true	
24	pairp X	false	X not of the form (pair A B)
25	sum n m	the sum: n + m	n and m both numerals
26	product n m	the product: n x m	↓
27	difference n m	the difference: n − m	:
28	quotient n m	the quotient: n ÷ m	:
29	less n m	the truth value: n < m	↑
30	greater n m	the truth value: n > m	n and m both numerals
31	and true X	X	
32	and false X	false	
33	or true X	true	
34	or false X	X	
35	not true	false	
36	not false	true	
37	if true X Y	X	
38	if false X Y	Y	
39	equals X X	true	
40	equals X Y	false	X and Y distinct constants
41	equals FA GB	and (equals F G) (equals A B)	F, G both constructions
42	and (or A B) C; and C (or A B)	or (and A C) (and B C); or (and C A) (and C B)	
43	and (and A B) C	and A (and B C)	
44	or (or A B) C	or A (or B C)	
45	and A (and (some n Q) B)	and (some n Q) (and A B)	A an atom
46	$\exists x_1 \ldots x_m$A	A	A is one of true, false
47	$\exists x_1 \ldots x_m$(or A B)	or ($\exists x_1 \ldots x_m$ A) ($\exists x_1 \ldots x_m$B)	
48	$\exists x_1 \ldots x_m$(and (some n Q) B)	$\exists x_1 \ldots x_m z_1 \ldots z_n$(and (Q $z_1 \ldots z_n$) B)	
49	$\exists x_1 \ldots x_m$C	$\exists x_1 \ldots x_{i-1} x_{i+1} \ldots x_m$C'	C is a conjunction with one of (equals x_iT), (equals T x_i) as a conjunct, and C' is C with T for x_i everywhere

FIGURE 1

The SUPER machine, underneath all its various trappings, is just a redex remover. Its repertoire consists of the 49 redex patterns shown in Figure 3, each one with its own characteristic contraction. The SUPER machine's contraction algorithm is simple to state but complicated to perform:

to obtain E_{i+1}, identify <u>all</u> redexes in E_i and contract them.

This is known as <u>full contraction</u>, and represents the maximum rate of reduction that is logically possible.

To explain the SUPER language, then, we must discuss these contractions and describe their intended joint effect.

Delta-contractions. To begin with, there are many redexes typified by arithmetical and logical expressions such as

sum 54 653, **and true false**,

whose corresponding contracta are constant symbols; in this case,

707, **false**,

by contractions 25 and 31, respectively. These redex patterns are all traditionally classified as "delta-redexes" (this greek-letter terminology is due to Curry).

The general idea is that a delta redex is a 0-ary combination

$$\alpha^n F C_1 \ldots C_n$$

whose head F and arguments $C_1, \ldots, C_n$ are all constant symbols, and that there is an associated algorithm to construct a constant symbol C as its corresponding contractum.

Constructors and data structures. Expressions in normal form are by no means always single symbols. The example of traditional numerical computation (which usually does end in a numeral or a truth value, the "result" of the computation) is not typical. For example, if we compute the inverse of a matrix, the result is a matrix, and this is an

expression more complicated than a single numeral, or scalar. Similarly, in general, a SUPER computation might end with an expression more elaborate than a constant. However, (by definition) the expression will not contain any redexes. This means that the combinations which it does contain need to be understood [illegible] as indications that, intuitively, some function [illegible] applied to some arguments.

The common thread throughout a reduction computation is that each [illegible] expression denotes the same object: all we are doing in carrying out the computation is, so to speak, exchanging one name of that object for another one. Some names are more elaborate than simple constant symbols, for example, the list whose first element is 6 and whose second element is 23 is in sugared form the expression

[6 23]

and is in raw SUPER the combination

α^2 **pair** 6 α^2 **pair** 23 **nil**

which contains no redexes. It is a data-structure. The 2-ary constant **pair** is therefore called a constructor rather than a functor, which is no more than to say that expressions of the form

α^2 pair C D

are not redexes, or to put it another way, that [illegible] the same [illegible] sum, product, difference, quotient, less, greater, and, or, if, [illegible] views [illegible] [illegible] [illegible] [illegible] together with [illegible] aspects of all the basic relationship of LNF to SUPER.

SUPER computation = getting rid of all SUPER redexes. To complete this overall view of the SUPER language, we now must look more closely at the other kinds of redexes which must be absent from an expression in order that it should be SUPER-normal, namely, those of contractions 41 through 49. It is these remaining contractions which

constitute much of the novel aspect of the SUPER language. In order to understand them we must next examine in more detail the structure and meaning of Horn predicates.

Goal clauses and Horn predicates. A goal clause is a sentence of the form

$$\exists x_1 \ldots x_k C$$

where $k \geq 0$, the x_i are distinct variables, and C is a conjunction whose conjuncts (if it is nonempty) are all atomic sentences. (When $k = 0$ then we omit the symbol $\exists$).

A Horn predicate is a predicate of the form

$$\lambda(z_1 \ldots z_k)\ D$$

where the z_i are distinct variables and D is a disjunction whose disjuncts (if it is nonempty) are all goal clauses.

For example, the following is a Horn Predicate:

```
λ(a b c) (or  (and (equals a nil) (and(equals b c) true))
         (or  ∃ x y w (and (equals a (pair x y))
                           (and (equals c (pair x w))
                                (and (append y b w) true)))
              false)).
```

SUPER's equivalent of PROLOG's Horn clause definitions. In the SUPER system, in order to define a k-ary constant R as standing for a k-ary relation in the manner of PROLOG or LOGLISP, one asserts a "Horn equation"

$$R = \lambda(z_1 \ldots z_k)[\ G_1 \textbf{ or } \ldots \textbf{ or } G_n\ ].$$

(In LNF-Plus one gets the same effect by a different transaction: see **Volume 3, 3.2**

Horn Clauses for how it is actually done at the LNF-Plus terminal).

Each disjunct G_i on the right hand side of the Horn equation is a goal clause of the form

[illegible] and [illegible]

[illegible]

[illegible] and $B \leftarrow A_1, \ldots, A_k$ and

namely one consisting of a conjunct which is an atomic sentence

$$R(A_1, \ldots, A_k)$$

whose predicate is the relation symbol R which has been defined by the above Horn

equation. (B and C are conjunctions of atomic sentences).

Then the contraction of the redex R to **R**, followed by the further contractions which "apply **R** to its arguments", transforms the goal clause (*) to

$$\exists y_1 \ldots y_n \, (B \text{ and } (G'_1 \text{ or} \ldots \text{or } G'_n) \text{ and } C) \qquad (**)$$

where G'_i is the result of substituting $A_1 \ldots A_k$ for $z_1 \ldots z_k$ in G_i, $i = 1, \ldots, n$. Note that (**) is no longer a goal clause: however, it will be transformed to a disjunction of goal clauses under the pressure of the SUPER contractions 42 - 48. Namely, contraction 42 will "push **and**'s through **or**'s" to cause (**) to become

$$\exists y_1 \ldots y_n \, (\, (B \text{ and } G'_1 \text{ and } C) \text{ or } \ldots \text{ or } (B \text{ and } G'_n \text{ and } C) \,)$$

and contraction 47 will "push $\exists$'s through **or**'s" to cause it to become

$$\exists y_1 \ldots y_n (B \text{ and } G'_1 \text{ and } C) \text{ or } \ldots \text{ or } \exists y_1 \ldots y_n \, (B \text{ and } G'_n \text{ and } C).$$

Finally, the existential quantifier prefixes of each G'_i will be pulled outside the and contractions 45 and 48, to yield

$$\exists y_1 \ldots y_n w_1 \ldots w_{k1} (B \text{ and } H_1 \text{ and } C) \text{ or } \ldots \text{ or } \exists y_1 \ldots y_n \, w_1 \ldots w_{kn} (B \text{ and } H_n \text{ and } C) \qquad (***)$$

where G_i is $\exists w_1 \ldots w_{ki} H_i$. The expression (***) is a disjunction of goal clauses.

This overall transformation of (*) to (***) corresponds to using each clause in turn in the PROLOG or LOGLISP definition of R to <u>eliminate a goal</u> from (*) when it is an atomic sentence $R(A_1 \ldots A_k)$.

We next discuss how to eliminate a goal in a goal clause when it is an equation. It is this kind of goal predicate which we <u>introduce</u> when we perform goal elimination in the way just described.

The elimination of equations from goal clauses. We consider a goal clause having one of the forms

Example

```
append = λaλbλc
          (or  (and (equals a nil) (and(equals b c) true))
          (or  (some 3 λxλyλw (and (equals a (pair x y))
```

(and (equals c (pair x w))
(and (append y b w) true))))

false))

The introduction of this definition corresponds to asserting the pair of Horn clauses

$\forall$x((**append nil** x x) $\leftarrow$)
$\forall$x,y,z,w((**append** (**pair** x y) z (**pair** x w)) $\leftarrow$ (**append** y z w)).

The definiens reduces to the expression

S S (C(B S)(B (B or) and (C equals nil)) (C (C and) equals true))
(C (C B (B (B (some 3)) S(C (B B (B (S B)) (B and) equals) pair))
((C (C B (C (B (B S) (B and) equals) pair)))
(C (C (C and))(C append) true))

which we will abbreviate as **APPEND**. Thus we have introduced a new contraction (with no effect: but see below), whose redex is the constant "**append**" and whose contractum is the expression **APPEND**. This new contraction will cause any atomic sentence of the form

(**append** a b c)

to be transformed first into

(**APPEND** a b c)

and thence into

(**or** (**and** (**equals** a **nil**) (**and**(**equals** b c) **true**)) (**or** (**some** 3 **E**) **false**))

where **E** is

(S (B (S B) (B and (equals a)) pair)
(C B (B S (B and (equals c)) pair)
(C (C and) (C append b) true)))

which is the normal form of the expression

λxλyλw (**and** (**equals** a (**pair** x y))
(**and** (**equals** c (**pair** x w))
(**and** (**append** y b w) **true**))).

Thus the atomic sentence (**append** a b c) becomes a disjunction with two goal clauses as disjuncts.

In fact a user definition does not cause an extra computation to be carried out. The [illegible] discussion was a necessary digression [illegible] the system actually [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

setof (p q) (**and** (**append** a b p) (**pair** 1 (**pair** 2 nil)) **true**)

[illegible] append [illegible]

setof (p q) (**or** (**and** (**equals** p nil) (**and** (**equals** q (**pair** 1 (**pair** 2 nil))) **true**))
(**or** (**and** (**equals** p (**pair** 1 nil)) (**and** (**equals** q (**pair** 2 nil)) **true**))
(**or** (**and** (**equals** p (**pair** 1 (**pair** 2 nil))) (**and** (**equals** q nil) **true**)) **false**

which can then be sugared for output as, say, the explicit set expression

{ ([] [1 2]), ([1] [2]), ([1 2] []) }

or whatever other surface syntax may be preferred. As can be seen, the final raw SUPER expression contains no redexes and is a "solved" Horn predicate: one whose disjuncts contain no existential quantifiers and whose goals are equations giving the components of each "solution". The use of the **setof** notation instead of the λ-notation is to prevent contractions 1 though 7 from transforming the variables p and q away and losing the logical structure in terms of which the logical contractions achieve the final solved form.

In the next chapter we examine in more detail how the contractions are effected in the graph representations of SUPER expressions.

CHAPTER 2. GRAPH REDUCTION.

In the SUPER machine an expression is internally represented by a graph. A graph is a system of nodes, all accessible from the root node of the system. Each node has a unique address, and a very small memory which can contain the addresses of other nodes. A node B is accessible from a node A if either A is B, or (recursively) if some node accessible from A contains the address of B. We shall make informal diagrams of graphs and indicate that a node A contains the address of node B by drawing an arrow from A to B.

Symbols. The representation of a symbol is a graph consisting of a single node which which contains information specifying its arity, its kind (variable or constant, relational or functional), and which particular symbol of that kind it is.

Combinations. The root node of a combination is labelled by the greek letter α and contains two addresses, an operator address which is directly or indirectly that of the root node of its operator, and an operand address which is directly or indirectly that of its operand.

Pointers In the definitions of combination and abstraction graphs we spoke of an address being "directly or indirectly" that of the root node of an expression. That was because in addition to symbols, combinations and abstractions we have pointers. A pointer contains the address of the node which is its target. Pointers permit a system of indirect addressing in graphs. A node directly addresses a node C if it contains the address of C, and indirectly addresses C if it contains the address of a pointer the target of which is C or indirectly addresses C. In diagrams of graphs a pointer is represented as a node labelled by the sign ∇.

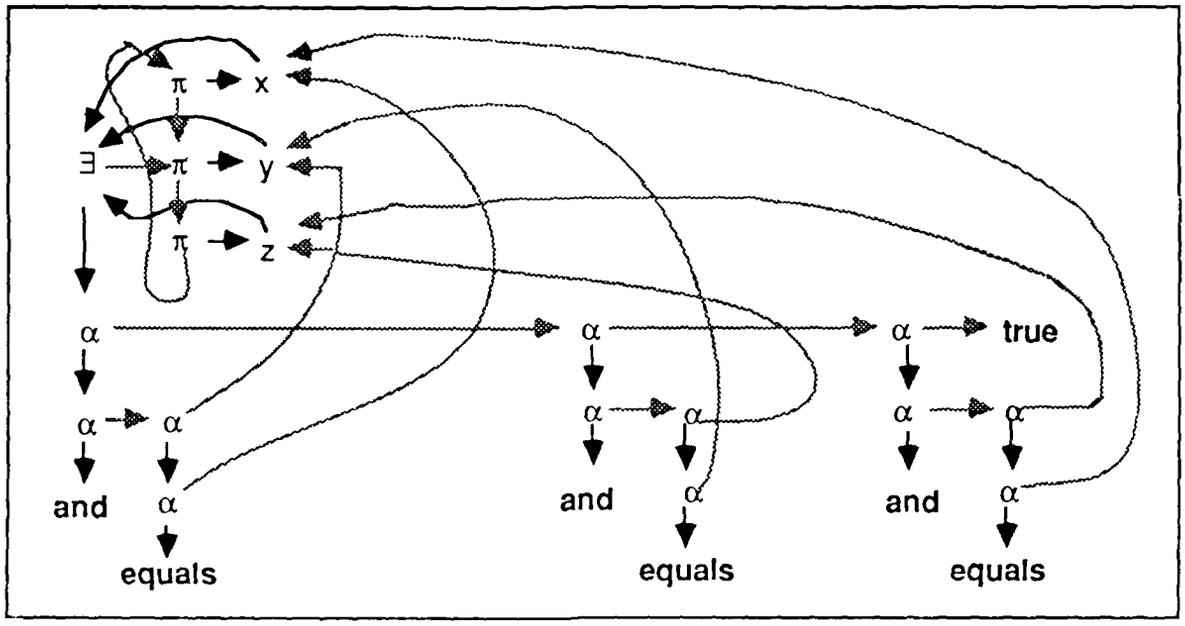
Existential quantifications. Prefixes. Quantifiers. Variables. An existential quantification is represented by a graph whose root is labelled by the existential quantifier sign ∃, and contains a matrix address, which directly or indirectly addresses the root node of the matrix of the quantification, and a prefix address. The prefix consists of n quantifier nodes, each of which addresses a distinct variable in the prefix of the quantification, and the prefix address is the address of an arbitrary one of these nodes. These quantifier nodes are linked into a ring: each quantifier node addresses the next one in the ring. A quantifier node is labelled by the greek letter π. (If n = 0 the prefix is empty, and the prefix address is **null**. The whole quantification expression is then equivalent to its matrix). Each variable node addresses the root node of the quantification, and corresponds to a distinct variable in the prefix of the quantification. For example, the graph of the quantification

∃ x y z B

has three quantifier nodes, one for each of x, y and z. Finally, throughout the matrix of the quantification, an occurrence of one of the variables is simply the address of the variable node corresponding to that variable. *For example, the existential* quantification

∃x y z (**and** (**equals** x y) (**and** (**equals** y z) (**and** (**equals** x z) **true**)))

is represented by the graph



The operand address of a combination cell is represented by a light arrow, and the operator address by a dark arrow. Each quantifier node is labelled by π and addresses its corresponding variable node by a dark arrow, and the next quantifier in the ring by a light arrow. The quantification node addresses its matrix by a dark arrow and (some node in) its prefix by a light arrow. Each variable in the prefix is actually anonymous - its distinct address is what gives it its unique identity. Thus, any other three variables would do just as well in the diagram: this corresponds to the well known fact that, in the ordinary linear notation, bound variables can be (with suitable safeguards to avoid clashes and captures) relettered without changing either the meaning or indeed the deep syntactic structure of the expression.

Motivation for the quantification representation. The reader may be puzzled at this point to know why we have a second, special, graph representation for existential quantifications. Is not the combination (**some** 3 λxλyλzB) the "official" expression, and is not the notation ∃xyzB merely sugar for it? Yes: nevertheless, inside the SUPER machine we use both representations.

The reason is that in the SUPER machine we implement contractions as local operations carried out by small "nodule" processors, one for each node of the expression graph; and in order to get correct global behavior we need two different (but semantically equivalent) syntactic representations of existential quantifications: one in which bound variables are abstracted away in favor of combinators (the **some** representation), and one in which they are not (the ∃ representation). Contractions 45 through 49 are designed to exploit this dual representation.

Contraction 49. According to contraction 49, if the matrix C of a goal clause

$$\exists x_1 \ldots x_n C \qquad (*)$$

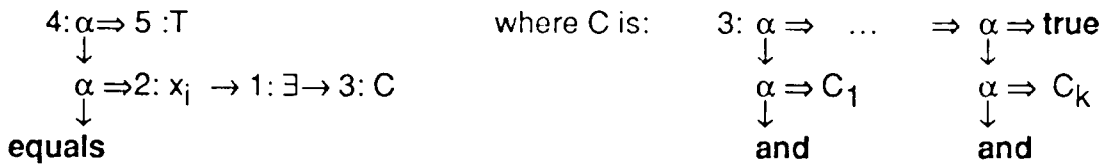
contains an equation (**equals** x_i T) or (**equals** T x_i) as a goal, then (*) should be contracted to the goal clause

$$\exists x_1 \ldots x_{i-1} x_{i+1} \ldots x_n C'$$

where the new matrix C' is the result of substituting T for x_i throughout C and the new prefix lacks the variable x_i. Now on the face of it this is a very nonlocal operation. There may be many occurrences of x_i in C and the substitution operation must cause each of them to become an occurrence of T. However, if we use the graph representation just described, contraction 49 can be effected in a surprisingly local and economical way.

It is not just the execution of the contraction which has a global character. The detection of the 49-redex pattern (which must also be done by the nodule) also appears to be a global process. The nodule playing the role of the root node of a 49-redex can be arbitrarily far away (in terms of addressing chains) from the equation, and it is impossible (as far as we can tell) to contrive suitable local operations which would detect the fact that it is a 49-redex and would then bring about the necessary changes. Our solution is to locate the center of the action not in the root nodule of the redex itself but in the root nodule of the equation.

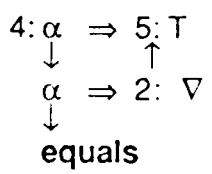
Consider the immediate "neighborhood" of the equation from the point of view of its root node:



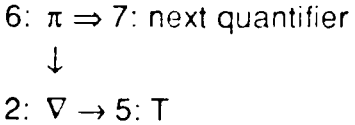
(we suppose that the address of the 49-redex is 1, that of the variable node corresponding to x_i is 2, that of the matrix is 3, that of the equation is 4, and that of the expression T is 5).

The nodule at 4 detects that it is the equation of a 49-redex located at 1 as follows. It detects that it is an equation simply by seeing the constant **equals** two links down the operator address chain; that one side of itself is a variable in the prefix of an existential quantification in whose matrix it is itself a subexpression, by looking at nodes 2 and 1; and that moreover the matrix is a conjunction of which it is itself one of the conjuncts, by having just received an ∃(1) message from node 1 as described immediately below.

The entire 49-contraction is then effected by the following <u>local</u> change:



Namely, the variable node associated with x_i is changed to a pointer whose target is the expression T. The quantifier node corresponding to x_i immediately notices that it is now addressing a pointer instead of a variable and accordingly changes itself to a pointer addressing the next quantifier. In graphical form, this quantifier excision thus consists of the quantifier node at (say) 6 changing itself from



to

6: $\nabla \rightarrow$ 7: next quantifier

2: $\nabla \rightarrow$ 5: T

whereupon the previous quantifier (the one addressing 6) will immediately bypass 6 and directly address 7 (see the discussion of such pointer-bypassing behavior in the next chapter).

Note that this can result, when n = 1, in the only remaining quantifier excising itself, that is, changing itself into a self-referential pointer (the unit prefix shrinks to a null prefix). In this case, the $\exists$-node will find itself addressing a null prefix and will change its prefix address to **null**.

Since every occurrence of the variable x_i throughout the matrix C directly addresses node 2, and since node 2 now has T as its target, each such place now indirectly addresses the expression T instead. The effect is that of a simultaneous substitution in constant time, regardless of the number of occurrences of the variable.

Thus the $\exists$-representation facilitates the 49-contraction enormously. It even permits simultaneous 49-contractions of the same goal clause with respect to two or more equations.

In the next chapter we discuss in more detail the contraction behavior of the nodule processors which comprise the bulk of the SUPER machine. We shall then see how an equation nodule can be made to "fire" a 49-contraction only when the context is indeed a 49-redex. In our example, node 4 must know that it is in fact one of the conjuncts C_j. As we shall see in the next chapter, it will knows this when, but only when, it receives an "$\exists$(1)" message from node 1 which has propagated to it along the spine of the conjunction. Intuitively, this message says to the equation that it is a goal in a goal clause whose root is node 1. Since the equation can see that its variable belongs to node 1, the message assures it that any variables in T which are also in the prefix of 1 will not be moved outside their scope by the substitution. This prevents, for example, the invalid contraction, fired erroneously by the equation x = y with respect to x:

$\exists$x(**and** (R x) (**and** $\exists$y(**and** x = y **true**) **true**) $\Rightarrow$ (**and** (R y) (**and** $\exists$y(**and** y = y **true**) **true**))

which brings the variable y out of its scope, but allows the correct contraction

$\exists x($ **and** $(R\ x)$ (**and** $\exists y($**and** $x = y$ **true**) **true**) $\Rightarrow \exists x($**and** $(R\ x)$ (**and** (**and** $x = x$ **true**) **true**))

to be fired correctly by the same equation with respect to y. The ∃-message can reach the equation from the inner quantifier but not from the outer one, since only **and**-nodes can pass such messages along to their arguments. An **and**-node is the root of a graph of the form

$$\begin{array}{l} \alpha \Rightarrow B \\ \downarrow \\ \alpha \Rightarrow A \\ \downarrow \\ \textbf{and} \end{array}$$

and it will propagate any ∃-message it receives both to A and to B. No other kind of node will propagate an ∃-message.

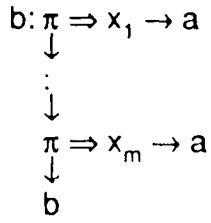
Contraction 48. This is the contraction in which we convert from the **some** representation of an existential quantification to the ∃-representation. A 48-redex has the form

$\exists x_1 \ldots x_m($**and**(**some** n Q) B)

where $m \geq 0$ and $n \geq 1$. Thus the expression Q is n-ary. The root of the redex is an ∃-node with matrix the **and**-node whose arguments are (**some** n Q) and B. In graphical form the redex is

$$\begin{array}{llll} a: \exists \Rightarrow & \alpha \Rightarrow & B & \\ \downarrow & \downarrow & & \\ b: X & \alpha \Rightarrow & \alpha \Rightarrow e: Q & \\ & \downarrow & \downarrow & \\ & \textbf{and} & \alpha \Rightarrow n & \\ & & \downarrow & \\ & & \textbf{some} & \end{array}$$

where X is the prefix:



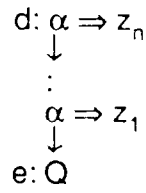
The 48-contractum is then obtained in two steps. First, node a executes the **allocate** instruction

c, d := (**allocate** n a e)

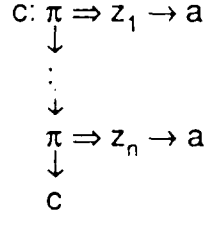
which causes the allocation of 3n new nodes, organized into the prefix and combination described below, and returns their addresses, c and d, respectively.

For convenience of the discussion, we shall suppose that the variables in the newly allocated prefix are $z_1 \ldots z_n$.

The newly allocated combination is (Q $z_1 \ldots z_n$), *that is, in graphical form*:



and the newly allocated prefix Z is, in graphical form:



that is, its variables all belong to (and therefore address) the calling node at a.

Next, the old and new prefixes X and Z are welded into a single prefix Y by interchanging the two NEXT addresses in nodes b and c, so that Y is the prefix:

b: $\pi \Rightarrow x_1 \rightarrow a$
$\downarrow$
$\pi \Rightarrow z_2 \rightarrow a$
$\vdots$
$\downarrow$
$\pi \Rightarrow z_n \rightarrow a$
$\downarrow$
$\pi \Rightarrow z_1 \rightarrow a$
$\downarrow$
$\pi \Rightarrow x_2 \rightarrow a$
$\downarrow$
$\vdots$
$\downarrow$
$\pi \Rightarrow x_m \rightarrow a$
$\downarrow$
b

and the new redex is formed:

a: $\exists \Rightarrow \alpha \Rightarrow B$
$\downarrow \quad \downarrow$
b: Y $\quad \alpha \Rightarrow d$
$\downarrow$
and

The net effect is to achieve the inference step

$$\exists x_1 \ldots x_m (\textbf{and}\ \underline{\exists z_1 \ldots z_n (Q\ z_1 \ldots z_n)}\ B) \Rightarrow \exists x_1 \ldots x_m z_1 \ldots z_n (\textbf{and}\ (Q\ z_1 \ldots z_n)\ B)$$

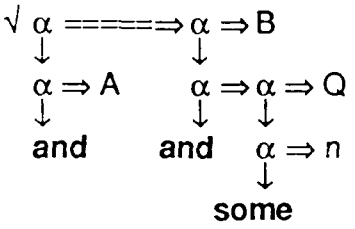
when the underlined part of the premise is actually represented as (**some** n Q). Recall that this inference is valid if and only if the variables $z_1 \ldots z_n$ do not occur free in B. Since they are in fact newly created by the allocation invoked in going from the **some** to the $\exists$ representation, this condition is automatically satisfied.

Note that the redex pattern calls for the **some** expression to be the leftmost conjunct in the matrix of an existential quantification. We thought about relaxing this condition and allowing it to be an arbitrary conjunct (the inference being still valid), but we were unable to deal satisfactorily with the nondeterminism introduced: a goal clause could then be a 48-redex in more than one way if its matrix contained more than one **some** expression.

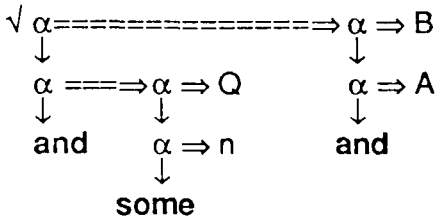
The normal form of the expression $(Q\ z_1 \ldots z_n)$ will in general have many occurrences of

each of the variables $z_1 \ldots z_n$. By our construction, these will all be direct references to the bound variable nodes, as is required by the representation.

Contraction 45. The purpose of this contraction is to move **some** expressions to the left within conjunctions, so as eventually to form 48-redexes. The 45-redex pattern in graph form is



and the 45-contractum is



with the condition that A be an atomic sentence (which prevents the loop which would be possible if A were itself a **some** expression).

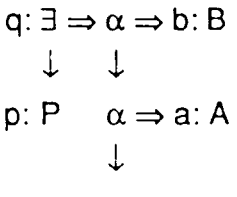
Contraction 46. This requires hardly any discussion. An existential quantification whose matrix is a truth value is equivalent to that truth value. In the graph representation, therefore, all that is required is for the root of the redex to change its label from ∃ to ∀ and to redesignate its matrix address as that of its target.

Contraction 47. This contraction embodies the simple basic inference in which an existential quantifier prefix may be validly distributed through a disjunction:

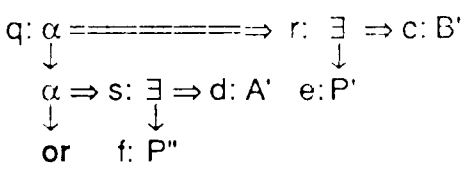
$$\exists x_1 \ldots x_m (\textbf{or } A\ B) \quad \Rightarrow \quad (\textbf{or } \exists x_1 \ldots x_m A\ \exists x_1 \ldots x_m B).$$

However, in SUPER we must implement this inference in such a way as to maintain the properties of the ∃ representation of existential quantifications. Recall that in the

redex each occurrence of a variable x_i in A or in B is simply the address of the corresponding variable node in the prefix. The 47-redex takes the graphical form



and the corresponding 47-contractum is



where r and s are the addresses of two nodes allocated by the **new** command, and the addresses c, d, e and f are those returned by the **copy** commands:

r, s := **new, new**
c, e := (**copy** p q r b)
d, f := (**copy** p q s a).

The command (**copy** p q r b) creates (1) a fresh copy, P', rooted at node c, of the prefix P rooted at node p and (2) a fresh copy, B', rooted at node e, of the expression B which is rooted at node b. However, wherever B refers to a variable x in the prefix P, B' refers instead to the corresponding variable in the new prefix P'. The variables in P' belong to the address r. The command (**copy** p q s a) creates P" and A' similarly, with addresses d and f.

This is where, in SUPER, the basic phenomenon of OR-parallelism arises: a variable may be bound in more than one way because there is more than one solution. In PROLOG the different values are encountered only sequentially, and therefore the same cell can be used for the variable, earlier values being overwritten by later ones. In LOGLISP the different values are encountered in quasi-parallel, and so there too the need arises for a different cell for each value (or what is the same, a different cell for each independent "copy" of the variable).

Global effect of combined local transformations. The plan behind these contractions is that if each nodule processor playing the part of a node in the graph representation of an expression repeatedly "fires when ready" according to its own perception of local information available to it, the combined global effect will be to reduce the whole expression, eventually, to normal form. If a nodule detects that the expression of which it is the root is the redex of some contraction, it will forthwith carry out the changes required to transform itself into the root of the corresponding contractum (assuming availability of the new nodule processors which it may have to allocate; see the discussion of garbage collection in the next chapter).

Some interesting questions are raised by this plan.

Will the contractions interfere with one another when performed concurrently? We believe they do not, but at present we do not see how to [illegible]. The following paragraphs are intended to be persuasive.

Referential transparency. The contractum and redex of each contraction are identical (they are to be the same object). Hence if we always see to it that the node which is the root of a contractum is the same node as was the root of the redex, we shall not affect the semantic correctness of any reference to that node from anywhere else in the graph. All addresses of the node will still, so to say, be referring to the same semantic object. This is what is meant by "referential transparency". Thus if, while a contractum is under construction at node i involving the addresses of one or more other nodes, these other nodes are themselves undergoing contractions, their addresses will remain semantically valid throughout the construction at node i. This relationship is symmetric: any other nodes addressing node i will likewise be semantically unaffected by the changes taking place at i.

It seems, therefore, that provided each contraction is carried out correctly in its own terms, the changes involved will indeed be referentially transparent , that is, invisible, to all other contractions. Existing structure does not disappear when contractions take place: it remains behind to be seen by other nodes who may still be sharing it. Nodes disappear only if, and when, they become inaccessible from the root node of the entire expression. Until it disappears, a node retains its semantically invariant significance. In the next chapter we consider in more detail the reclamation of inaccessible nodes.

Are the contractions locally deterministic? No expression can be a redex in more than one way. An apparent exception to this is the case of redexes of type 49. A 49-redex can be one with respect to two different equations, or indeed with respect to a single equation both of whose terms are variables in the prefix. The second case can be trivially resolved by taking one of the two variables (say, the left hand side of the equation) arbitrarily as the variable of the schema. The first case is more subtle. Since it is the root node of the equation, not the root node of the redex itself, which [illegible] 49-contractible, the non-determinacy is only apparent. The same goal clause will therefore be 49-contractible at the same [illegible] to more than one equations.

AND-parallelism in SUPER [illegible] AND-parallelism arises in the SUPER machine [illegible] unification algorithm. If two or more [illegible] same variable, there may very well be two or more simultaneous attempts to change the address in the corresponding variable node to (in general) different targets (that is, two or more different "bindings" for the same variable will be attempted). Only one of these, of course, can be allowed to succeed, and fortunately the logic of the transformation is indifferent to which of them it is. As we shall see in the next chapter, the reconciliation of conflicting messages is very easy to deal with in the Connection Machine on which we first intend to implement SUPER, and so in practice one of the "bindings" will be accepted and the others rejected without any further complications.

It is intrinsic to the full-contraction SUPER scheme that the relation symbols in all "Horn goals" in a goal clause get simultaneously replaced by the corresponding Horn predicate expressions, giving rise to AND-parallelism at the higher level of goal elimination.

CHAPTER 3. THE SUPER MACHINE

and the active nodular memory in which it can store a graph expression for parallel reduction processing. The nodular memory consists of a large number of small nodule memories, each one belonging to a small processor called a nodule. As far as the

front end machine is concerned the nodular memory is just a region of its total memory, which can be written to and read from, both in the usual one-word-at-a-time manner and also in a parallel, all-words-at-once manner. Finally, it has a Postal System, which is a high speed, high bandwidth communication network modelled on that of the Connection Machine, over which the nodules can send and receive messages to each other.

A SUPER user will interact directly only with the front-end machine, which will accept as input a (possibly sugared) SUPER expression D for reduction to normal form. The front-end machine translates D into the corresponding raw SUPER graph expression E and installs E in the nodular memory, where it is then reduced to normal form. Upon completion of the reduction process the resulting graph is then recovered from the nodular memory by the front-end machine and translated back to the sugared notation for output to the user. The translation by the front-end machine eliminates all abstractions by appealing to contractions 1 through 20. Thus the graph representation scheme which follows does not need to represent abstractions as such. However, as we shall soon see, the graph representation of quantifications indirectly, but in a highly controlled way, permits the graphical representation of the abstractions which occur as arguments to the **some** functor.

The whole transaction, from the user's viewpoint, feels just like a transaction with the single-processor LNF-Plus system. The work done is abstractly the same, and differs only in being done by steps of multiple redex contraction (indeed, ideally, full contraction) rather than single redex contraction.

The nodular memory: 2^n nodules linked by the SUPER Postal System. The main feature of the SUPER machine is its nodular memory, which instead of being a collection of passive storage cells each having no capability other than to hold a unit of information for later retrieval, consists of a large number of active elements, called nodules. Each nodule is a small special-purpose computer which can read from and write into its own local memory and can also do simple computations. The SUPER

machine's modular memory consists of the combined local memories of all the [illegible]

[illegible]

[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
UNKNOWN	ACTIVE	FREE
ACCESSIBLE	LNF	

The 1-bit fields in the first group of six are known as type fields. Exactly one of the type fields is set to **true** and the rest to **false**, in order to show which type of node the nodule is representing. The remaining five fields are for various housekeeping purposes which will be discussed later. Each nodule's remaining memory depends on the contents of its type fields.

A combination nodule (one whose COMBINATION field is **true**) has two n-bit fields, OPERATOR and OPERAND.

A quantification nodule (one whose QUANTIFICATION field is **true**) has two n-bit fields, MATRIX and PREFIX.

A quantifier nodule (one whose QUANTIFIER field is **true**) has two n-bit fields, CONTENTS and NEXT.

A constant nodule (one whose CONSTANT field is **true**) has one n-bit field, WHICH.

A pointer nodule (one whose POINTER field is **true**) has one n-bit field, WHERE.

A variable nodule (one whose VARIABLE field is **true**) has one n-bit field, WHOSE.

Thus we have nodules with two different sizes of memory. The larger memory size is that of the combination, quantification, and quantifier nodules, which need two n-bit fields in addition to their type fields and housekeeping fields. The smaller memory size is that of the pointer, variable and constant nodules, which need only one n-bit field in addition to their type field and housekeeping fields.

We refer to the COMBINATION field of the combination nodule whose address is i as COMBINATION[i], and similarly for other fields and other addresses.

A λ-free (graph) expression E is installed by the front end machine in the nodule memory in the following way. Each node in the graph is assigned a [illegible] address i. Then:

(1) If the node is the root of a combination, the field COMBINATION[i] is set to **true** and all the other type fields are set to **false**. The fields OPERATOR[i] and OPERAND[i] are then set respectively to the addresses of the operator and the operand.

(2) If the node is a constant, the field CONSTANT[i] is set to **true** and the other type fields are set to **false**. The field WHICH[i] is set to the bit pattern corresponding to the constant.

(3) If the node is a pointer, the field POINTER[i] is set to **true** and the other type fields are set to **false**.The field WHERE[i] is set to the address of its target.

(4) If the node is a quantification, the field QUANTIFICATION[i] is set to **true** and the other type fields are set to **false**. The field MATRIX[i] is set to the address of the matrix of the quantification. If the bound variables of the quantification are [illegible] through [illegible] and [illegible] through [illegible], then the fields VAR[illegible] are set to **true** and the fields [illegible] particular choice among [illegible]

The REFERENCE-COUNT field of each node is initialized to the correct value by the front-end machine when the initial graph is installed in the nodule memory.

The behavior of a SUPER nodule. When the nodules collectively are set up to represent in this way the graph expression E, they can be given their signal to start their reduction behavior.

The main mission of the ith nodule is to find out if node i is the root of a redex and, if so, to bring about the changes which represent the corresponding contraction. As the global reduction process continues, each node in the graph experiences a succession of redex-contraction changes, and it is the job of the nodule representing that node to make those changes.

The auxiliary mission of the ith nodule is to manage the propagation of certain information about its own status with respect to the reduction process as it unfolds. This involves the continual monitoring of both its own status fields and those of its immediate descendants, i.e., LNF, ACCESSIBLE, ACTIVE, UNKNOWN, FREE and REFERENCE-COUNT, so as to be ready to take the correct action when the appropriate circumstances arise, as discussed below. This auxiliary behavior is not strictly speaking reduction behavior: it is more in the nature of housekeeping behavior.

The reduction behavior of nodules. It helps the intuition to imagine oneself cast in the role of a nodule, and to go through the different possible circumstances which can arise locally, together with the appropriate actions. To begin with, there are many circumstances in which doing nothing is the correct action. For example, if our REFERENCE-COUNT is 0 then we are an inaccessible vertex, and we simply wait for something to happen (actually, as will be discussed below, since this means that we are a "free" nodule, available to be allocated when some contraction elsewhere in the graph requires a new node, we can in fact do something useful, namely propagate the news of our inaccessibility to our immediate descendant(s) by subtracting 1 from their reference counts, **null**ifying our pointers to them, and then setting our FREE flag to **true**). More generally, we do nothing unless we perceive that some action is actually called for. However, our passivity is busy since we must remain watchful, continuously monitoring the various information fields in our "neighborhood". When any of them

change it may be necessary to spring into action.

The neighborhood of a nodule. We assume (as is the case immediately after the graph has been set up in the nodular memory by the front-end processor) that the type fields in each nodule always satisfy the condition that exactly one of them is **true** and the rest **false**. We shall say that a nodule is a combination, etc., if its COMBINATION field is **true**, etc.

Constants are completely passive. The job of a constant is to stay entirely inactive. It is there to be addressed by others, and that is all it needs to worry about except (see above) to watch its own reference count in order to be ready to return itself to the pool of free nodules when the reference count goes to 0.

Pointers do very little. The job of a pointer at address i is just to point at another node WHERE[i], its target. This is rather an undemanding role except that its target may suddenly itself become a pointer, in which case WHERE[i] should be changed to WHERE[WHERE[i]]. The point of this change, no pun intended, is that pointers should always be bypassed whenever possible as part of the ongoing compaction of the graph. This is all a pointer node i has to worry about, but to do this properly the pointer i must monitor two fields. First, it must watch the field

POINTER[WHERE[i]]

in its target in order to detect when it changes to **true**. This will happen for example if the node WHERE[i] undergoes an I- or K-redex contraction. The moment that node WHERE[i] becomes a pointer, its WHERE field becomes the address of its target. So the second field that pointer i must monitor is the WHERE field of its target

WHERE[WHERE[i]]

ready for the moment when the action

(1) increment REFERENCE-COUNT[WHERE[WHERE[i]]]
(2) decrement REFERENCE-COUNT[WHERE[i]]
(3) replace WHERE[i]
by WHERE[WHERE[i]]

must be taken. In order to perform this monitoring function correctly, the nodule playing the part of pointer i must read from POINTER[WHERE[i]] and WHERE[WHERE[i]] once each message cycle, and execute this action each time that POINTER[WHERE[i]] is true.

Combinations do a lot of work. A combination i has to be ready to spot that it has suddenly become a redex and to take the appropriate action. Consider the possible "views" which a combination has of its neighborhood. First, no matter what kind of redex it is, or even whether it is a redex or not, a combination must take special action if its operator or operand is a pointer, namely, bypass that pointer just as we have already explained. Thus, if i "sees" that

POINTER[OPERATOR[i]]

is **true**, it knows that the node OPERATOR[i] is a pointer and that what it must do is the action

(1) increment REFERENCE-COUNT[OPERATOR[WHERE[i]]]
(2) decrement REFERENCE-COUNT[OPERATOR[i]]
(3) OPERATOR[i] becomes OPERATOR[WHERE[i]]

which bypasses that pointer. Similarly for its operand. We shall from now deal with combinations which do not have an operator or an operand which is a pointer.

Let us discuss a few of the contractions to get the general idea of the contraction behavior of a nodule. First, we consider contraction 18, which deals with **I**-redexes.

Contraction of the I-redex. If nodule i is an **I**-redex what it sees is that

CONSTANT[OPERATOR[i]] **and** WHICH[OPERATOR[i]] = **I**

and what it must therefore do is the following

(1) decrement REFERENCE-COUNT[OPERATOR[i]]
(2) negate COMBINATION[i] and POINTER[i] (turn itself into a pointer)
(3) set WHERE[i] equal to OPERAND[i] (to its former operand)

If the WHERE field of a pointer nodule is the same physical piece of memory as the OPERAND field of a combination nodule, then step (3) is automatic.

We next consider contraction 17, which deals with **K**-redexes.

Contraction of the K-redex. Nodule i detects that it is a **K**-redex by seeing that the fields COMBINATION[OPERATOR[i]] and CONSTANT[OPERATOR[OPERATOR[i]]] are both **true** and that WHICH[OPERATOR[OPERATOR[i]]] = **K**. What it must then do is the following

(1) negate COMBINATION[i] and POINTER[i] (turn itself into a pointer)
(2) set WHERE[i] equal to OPERAND[OPERATOR[i]] (to the operand of its former operator)
(3) decrement REFERENCE-COUNT[OPERATOR[i]]
(4) decrement REFERENCE-COUNT[OPERAND[i]]
(5) increment REFERENCE-COUNT[OPERAND[OPERATOR[i]]]
(6) set OPERATOR[i] and OPERAND[i] to **null**.

Contraction 14 : the pure S-redex. Nodule i detects that it is a 14-redex by seeing that the fields

COMBINATION[OPERATOR[i]]

COMBINATION[OPERATOR[OPERATOR[i]]

CONSTANT[OPERATOR[OPERATOR[OPERATOR[i]]]

are all **true**, that

WHICH[OPERATOR[OPERATOR[OPERATOR[i]]]

is **S**, and that if all of

COMBINATION[OPERAND[OPERATOR[OPERATOR[i]]]

COMBINATION[OPERATOR[OPERAND[OPERATOR[OPERATOR[i]]]]

CONSTANT[OPERATOR[OPERATOR[OPERAND[OPERATOR[OPERATOR[i]]]]]

are **true** then

OPERATOR[OPERATOR[OPERAND[OPERATOR[OPERATOR[i]]]]]

is not **B**.

Let

f = OPERAND[OPERATOR[OPERATOR[i]]]

g = OPERAND[OPERATOR[i]]

x = OPERAND[i]

What nodule i must do is the following:

(1) a, b := **new**, **new**

(2) set up a and b to be combinations with reference counts of 1 each and with

OPERATOR[a] = f and OPERATOR[b] = g,

OPERAND[a] = OPERAND[b] = x.

(3) increment the reference-counts of f, g, and x

(4) decrement the reference count of OPERATOR[i]

(5) set OPERATOR[i] = a and OPERAND[i] = b.

The other contractions in the "combinator" group (8 through 20) are handled in

analogous fashion.

Contractions 1 though 7. These contractions are intended for execution only by the front-end machine, and raise no problems of principle.

Behavior of quantification nodes. A quantification node whose address is i has the responsibility to transmit to its matrix, once each message cycle, the message ∃(i). This message will be retransmitted immediately upon receipt by any node of the form (but of no other form)

$$\sqrt{\alpha} \Rightarrow B$$
$$\downarrow$$
$$\alpha \Rightarrow A$$
$$\downarrow$$
and

to the roots of A and B. As was explained in the previous chapter during the discussion of contraction 49, this ∃-message system is part of the engineering of the correct implementation of that contraction. Any equation (one of whose expressions is a variable) which receives such a message can immediately know whether to fire by comparing the address in the message with that of the owner of the variable.

A quantification node must also watch the node addressed by its PREFIX field, which will normally be a quantifier node. If (as we saw can happen) the quantifier node changes itself into a pointer node, then the quantification node will routinely bypass the pointer by changing its PREFIX field to address the target of the pointer. However, because of the ring structure of prefixes, it is possible that the target of the pointer may be the pointer itself - the null prefix case. In this case the normal bypassing procedure would merely reproduce the self-reference. Therefore, instead, the quantification node will change its PREFIX field to **null** .

Short average lifetime for nodes in graph reduction. In general one must imagine the creation of new nodes as going on continuously at the same time as the process by

which nodes become inaccessible and therefore available for reuse. The faster both of these birth and death processes go, the shorter the average lifetime of each node. If the births happen faster than deaths, the finite pool of available nodes will soon become empty. There is therefore a premium on the rapid detection of inaccessible nodes: ideally they would be reclaimed at the moment they become inaccessible, but of course the engineering reality is that this moment may pass unnoticed with the consequence that a (mathematically, objectively) dead node may go on living (and, compounding the problem, consuming resources) for quite some time before finally being detected, killed and recycled.

New nodes allocated at very high rate. Experience with single processor systems shows that new nodes are indeed created at a very high rate. Consequently, unless this is offset by an equally high or higher, rate of node recovery the computation will soon end when the pool of available nodes becomes empty. This is why we attach great importance in the design of our nodule processor to its role as scavenger.

A remark concerning nodule recovery. In the contraction of a 14-redex at node i the following could be the case (and similar remarks apply to other contractions). Let c, d, and e be the vertices

OPERATOR[i],
OPERATOR[OPERATOR[i]] and
OPERATOR[OPERATOR[OPERATOR[i]]]

Then it is often the case that the reference count of c is 1 - that is, that the only vertex pointing at c is i. In that case, c can be used as one of the new vertices a, b. Less often, it happens that not only is the reference count of c equal to 1, but also that of d. In this case, d may then be used as the other of a and b. Finally, the reference counts of all three vertices c, d and e may be 1. In that happy case, not only do we not need to draw new vertices from the pool, but we can even contribute one to it! This remark shows that we can often hope to speed things up by avoiding the formality of putting nodules

back in the pool only to draw them out again immediately. A similar remark applies, mutatis mutandis, to the contraction of the other types of redex.

Returning oneself to the pool of free nodules. There is a circumstance, already noted earlier, under which any node i can take a useful action even though it is not a redex. If the reference count of i is 0, and it is a constant, then it should do the following

(1) CONSTANT[i] is set to **false**
(2) FREE[i] is set to **true**. (proclaim its availability).

On the other hand, if it is a pointer, it should

(1) decrement the reference count of WHERE[i]
(2) set POINTER[i] to **false**
(3) set FREE[i] to **true** (proclaim its availability).

If i is a combination, it should

(1) decrement the reference counts of OPERATOR[i] and OPERAND[i]
(2) set COMBINATION[i] to **false**
(3) set FREE[i] to **true** (proclaim its availability).

Similarly for other types of node.

Nodule memory management. The nodule-consuming commands **new**, **allocate** and **copy** are executed by the nodules in SIMD fashion under the supervision of the front-end machine. We at present believe that the technique needed for their implementation is a straightforward generalization of that used in the Connection Machine operating system for the implementation of the command **new** alone. Namely, the front-end machine coordinates all simultaneous requests for free nodules, computes the responses and transmits them to the requesting nodules. Under the

assumption that the pool of free nodules is large enough to satisfy all requests, the extra complexity in the SUPER machine caused by the **allocate** and **copy** commands does not raise any new basic issue. It is simply that in the SUPER machine the new nodules will in general have to be organized into multi-nodule structures (as explained in the previous chapter) rather than supplied one at a time (as is all that **new** requires). The **copy** command is almost entirely analogous to the classic LISP command of the same name, but with nodule processors taking the part of **cons**-cells

This is a correct realization of full reduction That the above nodule behaviors constitute a correct realization of the abstract process of full reduction remains to be demonstrated. The main issue is the coordination of the actions of the nodules with respect to [illegible]. When a nodule i changes its local memory to reflect a redex contraction that it [illegible] the neighborhood of i, some of which of course will be involved in the contraction [illegible] are themselves (perhaps) also changing their own local information. What we need to show is that the correctness of the changes made by the nodule i is not vitiated by the changes made by those in the neighborhood of i.

If a nodule is a redex, then its spinal neighbors are not. The spinal neighbor of an **I**-redex is the constant **I**. The spinal neighbors of the **K**-redex are the combination **K**x and the constant **K**. The spinal neighbors of the redex **S**fgx are the combinations **S**fg and **S**f and the constant **S**. None of these is a redex, and moreover none will ever become a redex as long as its FREE field remains **false**. (Of course, after reclamation into the pool of available nodules, the next incarnation of the nodule may well be as a redex, but this is irrelevant). It follows, therefore, that a nodule which needs to read from the fields of its spinal neighbors in order to reset its own fields to represent the result of a contraction can rely on those fields to remain unchanged.

The condensation of increment and decrement messages. In order to continue the correctness argument, we need to know about one of the global capabilities of the SUPER machine. In particular, the SUPER Postal System (after the fashion of the Connection Machine's Communication Network) provides a <u>condensation</u> service for

"increment" and "decrement" inter-nodule messages. This works as follows: during any one delivery cycle, a nodule i will be, in general, the addressee of many such messages, say, m increments and n decrements. They will have been sent to i by all the other nodules which want to increment or decrement the reference count of i. Instead of delivering to i such a (possibly) huge bundle of messages the Postal System delivers one message: add the integer (m - n) to your reference count.

Nodules are small, simple machines allowing large scale replication. A local nodule memory needs only a small number of bits (80 bits is more than enough, assuming 32-bit addresses) to accommodate its various type, housekeeping and address fields. A nodule's state is essentially, then, just a single 80-bit word. However, in order to accomplish its various contraction tasks a nodule must take into account features not only of its own state but also of those of its neighbors. This information will be acquired via incoming messages and will need to be stored locally. So we must postulate a further, working region of a nodule memory big enough to accomodate the largest amount of such neighborhood information which might be required. This case arises with the 12-node neighborhood which is involved in the pattern of redex 45. If we set up enough space to store the state words of 12 other nodes we would need a total nodule memory of 13 x 80 = 1040 bits. This should be compared with the Connection Machine's 4096 bits per processor.

The nodule's processing logic is simple. It has to perform a 42-case analysis to detect which, if any, of the redex patterns it represents, and then must be ready to execute the appropriate contraction once a redex pattern is detected. (As we noted earlier, the first 7 of the 49 redexes will not arise in the graph representation). In addition to this cycle of redex-contraction patrolling, a nodule must test to see whether it needs to carry out the various propagation chores we have discussed, bypass any pointers it can see, and so on. The entire code for all of this amounts to no more than a quite short straightline program.

Thus the nodule is suitable for large scale replication, in the manner of the Connection

Machine's memory processors.

The allocation of new nodules from the pool. The pool of free nodules at any time is just the set of nodules whose FREE field is **true**. Also, in any one machine cycle, there will be a set of requests for new nodules, arising from various contractions which are taking place at that cycle and the consequent various **new**, **allocate** and **copy** requests. The SUPER front-end processor, via the Postal System, coordinates all such requests and (provided that the total number of new nodules required does not exceed the number of available nodules) satisfies each **new** request by sending a unique free nodule address to the requesting nodule, and modifies each **allocate** and **copy** request appropriately to the parameters of the request. These facilities generalize the **processor-cons** feature of the Connection Machine, as described by Hillis and Steele [17].

In the event that there are more nodules needed to satisfy all requests than there are free nodules, some requests must perforce be delayed according to some fair discipline whose details will be related to the general scheme for controlling the firing of contractions.

Global accessibility analysis. The front-end processor must periodically, in any case, initiate a global accessibility analysis in order to detect nodules which, although possessing a nonzero reference count, are inaccessible from the root nodule and hence ready to be returned to the pool. This process is essentially the same as the marking process in classical LISP-like garbage collection. It involves interrupting the activities of the nodules, telling them to **freeze** the addressing topology existing at that moment. First, the ACCESSIBLE and ACTIVE fields of every nodule are simultaneously set to **false**, except for that of the root nodule, in which the fields are set to **true**. Then the nodules are restarted in a special state in which each nodule i monitors its ACCESSIBLE field, watching for it to become **true**.

The moment this occurs, the following action is taken:

(1) the ACCESSIBLE and ACTIVE fields are set to **true** of the immediate descendants (if any) of whose ACCESSIBLE fields are **false**

(2) the field ACTIVE[i] is reset to **false**

(3) **stop**

The front-end processor will detect the termination of this accessibility analysis by the fact that all nodules have a **false** ACTIVE field. The above program, running on every nodule, guarantees that there is always at least one ACTIVE field **true** until the analysis is completed. At its completion every nodule has a **false** ACCESSIBLE field if, and only if, it is inaccessible from the root: at which time all such nodules set their FREE fields to **true**.

The SUPER front-end machine. The role of the front-end machine is to be the interface between the user and the "invisible" nodules and postal system. The user submits one or more definitions D and an expression E to the front-end machine. D and E are formulated in a sugared version of the language (whose details we do not concern ourselves with here), just as in the single-reduction system LNF-Plus. The construction of the graph representation G of E and of the graph representations of the definientia of D are carried out by the front-end machine, again just as in LNF-Plus. Each occurrence of the definiendum of a definition is translated to a reference to the root of the graph of the corresponding definiens.The processing of (the graph G of) E takes place entirely within the nodules and is the reduction of G to normal form that we have been discussing. Once the graph is in normal form, its translation back into string notation is carried out by the front-end machine. There is much freedom in this part of the system for friendly sugaring of expressions to suit the user's tastes, and none of the problems we have discussed are affected by how this (relatively) superficial part of the system is handled.

The whole transaction appears to the user simply as the evaluation of "E **where** D" and the parallel nature of the reduction is invisible.

The management of scarce resources. The front-end machine plays a crucial role in the allocation of new nodules to service the various **new**, **allocate** and **copy** requests issued as a result of non-conservative contraction processes. (Contractions 17 - 19, 25 - 34, 37, and 38 are conservative, in that they require no new nodes to be allocated). In coordinating such requests it must inevitably encounter the problem of an excess of demand over supply, and the consequent need to delay the servicing of some requests at the expense of others. This problem is just another version of the problem of controlling the otherwise uninhibited "explosion" of concurrent contractions: should every node fire as soon as it detects that it is a redex, or should it wait until it has permission to fire? Who decides, and what is the basis of the decision? The decision would appear to be a global matter, not a local one. No nodule can know enough to make the decision responsibly. It is, therefore, the front-end machine, with its global view of the whole graph at once, which must administer whatever policy of scarce resource allocation we can devise. But what should the policy be?

This is in fact a very hard and widely studied problem, and we do not pretend that we have discovered a solution to it. The whole point of parallel computation is that some subtasks should be started even though we are not sure at the time that they will actually be needed for the eventual output. For example, the classical conditional expression (**if** A **then** B **else** C) normally is evaluated by first evaluating A, in order to decide which of B and C to evaluate and thus complete the process. This means that the time needed for the whole task is at least: time(A) + min(time(B), time(C)). All opportunities to process the B or C parts of the task in parallel are foregone, on the grounds that some (but we do not know which part) of the work will have been wasted. Obviously, in order to gain time through concurrency, we have to give up the policy of avoiding useless work. In fact, if we gain speed by evaluating both B and C at the same time as A, how can we so easily say that the unused B (or C, as the case may be) represents useless work? After all, it gains us the speed.

So it is with the general problem of deciding which contractions to perform. The easy

policy of "fire when ready" assumes, in effect, infinitely many nodules. We do in fact have a lot of them, and in many smaller problems we can perhaps actually operate in this mode. (Some LISP computations require no garbage collection!)

Generalized breadth-first control. Probably what has to be done is to operate according to some generalized a priori notion of "fairness" so designed as to prevent any one node from falling too far behind in the development. For example, a simple scheme would be for each new nodule to be given a time-stamp recording the time at which it is allocated, and for nonconservative redexes to be contracted (if demand exceeds supply) according to an oldest-first priority discipline. We shall be able to experiment with various schemes of this sort when the Connection Machine becomes available later in 1987.

SIMD architectures. Computer architects distinguish two kinds of parallel architecture:

- **S**ingle **I**nstruction-stream, **M**ultiple **D**ata-stream (SIMD)
- **M**ultiple **I**nstruction-stream, **M**ultiple **D**ata-stream (MIMD).

In SIMD machines a sequence of intructions comprising a single program P is broadcast by a "master" processor simultaneously to many "slave" processors, each of which executes these instructions in lockstep with the other slaves. However, each individual slave acts upon its own data set, so that in general different computations take place in each slave: the same program running on different data. The outputs of each slave can be communicated not only to the master but also to its neighbors, according to whatever is the connection topology of the whole machine - say, the 2-dimensional grid topology in which each slave has four neighbors: North, South, East and West.

Such a grid organization is a natural one for solving many types of computation arising from the partial differential equations of physics and engineering. For example, in

solving the equation for the equilibrium distribution of heat over a region of a plane surface one represents the region as a grid of cells, each (except for those on the boundary of the region) with four neighbors. Each cell, represented by its own slave processor, repeatedly computes, as its own value, the number which is the average of the values of its neighbors. Given fixed values in the boundary cells, and some initial values in the interior cells, this computation can proceed through as many iterations as are needed to arrive at the equilibrium.

The Connection Machine is SIMD. The Connection Machine has a SIMD architecture but with the interesting difference that the connection topology [illegible] neighborhoods of the slave processors is not fixed once for all (by the way they are wired together) but is variable, under program control, because the [illegible] software connections rather than hardware connections.

Connection Machine's general purpose communication network. Of course the slave processors in the Connection Machine <u>are</u> physically wired together in a fixed pattern (as the vertices of an n-dimensional hypercube), but this "hardware-level topology" is <u>not</u> the topology with which the program is directly concerned. Rather, there is a "higher-level topology" which, in effect, is <u>simulated</u> by the connection hardware. This is based on the idea that the hypercube connections can be made to support a <u>message-passing</u> system in which, in one "delivery cycle", every slave processor can send a message (of fixed length, say, 32 bits) to, and receive a similar message from, every other slave processor. This capability is that of a <u>postal system</u> - one sends messages to a given address and receives messages sent to one's own address. The "soft" topologies which can be set up via such a postal system can be varied, under program control, and made to be whatever is appropriate for the problem at hand despite the fact that the "hard" topology on which it is running remains fixed by the pattern of the actual physical wiring interconnecting the slaves.

The topology can vary even during the computation. Not only can one set up the Connection Machine at the start of a problem to have, say, a grid topology, or a butterfly

topology, or whatever - in imitation of any one of the fixed-connection parallel machines - but by suitable programming one can even get the effect of being able to rewire the connections during the computation, changing the connection patterns to suit the developing demands of the problem as imposed by the data.

Data parallelism vs. control parallelism. This is particularly necessary in problems whose inherent parallelism resides in the data rather than the algorithm. In many problems the data objects consist of very large numbers of elements connected together in meaningful ways which are part of the data structure (as, e.g., in the expression graphs of our earlier discussion) and the parallelism in the computation consists of the concurrent transformations which take place at the different elements of the overall data structure. These typically (as in the expression graph case) result in changes in the data structure, old elements disappearing and new ones being created, and old connections being severed and new connections being introduced. The changes are brought about by the presence of the same "algorithmic force" being continuously felt throughout the data, as it were, just like one of the natural fields described by the physicists. One has, so to speak, to give the "equations of the force field" in the form of an algorithm which must be executed over and over again at each "point" in the "data space". This kind of computation exhibits what Hillis and Steele call data parallelism.

This is in contrast with control parallelism, which appears in computations where there are many different algorithms running "processes" which exchange data and "cooperate" with each other in order to accomplish some global objective. This is the most intricate sort of MIMD parallelism, the kind of parallelism which most people think of when parallel computation is mentioned - many different machines working at the same time, each doing its own thing but interacting suitably with the other machines from time to time. Control parallelism is very difficult to program, since the interactions between processes can be extremely complex. One has to program each machine separately, but with one eye on the programs for all the other machines, so that each machine deals properly with the interactions in which it may take part. In a sense one

must be "conscious" of the entire system at all times.

Data parallelism on the other hand is far easier to manage. We write only one (often quite simple) program which runs a process which is "cloned" into a horde of identical processes at work all over the data space. The interactions between these processes are typically far less complex than those which arise in control parallelism, and one can, so to speak, relegate them to the "unconscious" since they do not require conscious, explicit supervision. The interactions are automatic side effects of the "force field" which is imposed by the common algorithm at work throughout the data.

The Connection Machine can simulate the SUPER machine. Our design for the SUPER machine is a mixed SIMD-MIMD architecture. Our nodule processors have a certain amount of local autonomy and carry on with their own computations independently of the front-end machine. However, the front-end machine can, where necessary, interrupt the nodule computations with a **freeze** command and then assume control. In effect it can switch the system to SIMD mode from MIMD mode and then broadcast instructions to all the nodules simultaneously. This, it will be recalled, is how we do the "setup" phase of the accessibility analysis. As soon as the nodules are readied to propagate their accessibility, the front-end frees them to return to MIMD mode.

Hillis' and Steele's example. In their paper Data Parallel Algorithms Hillis and Steele use the process of parallel combinator reduction as one of several examples illustrating the way one can program the Connection Machine to take advantage of the natural data parallelism in a problem. We have edited their pseudo-ALGOL code somewhat to make it more intelligible, correct one or two minor errors, and render it more compatible with our own notations, but otherwise we have maintained the spirit of their illustration. The idea of the program should be quite clear in the context of our previous discussion. The function **new** returns the address of a new processor, in analogy with LISP's **cons**. Indeed Hillis and Steele call it **processor-cons**, rather than **new**.

It is not our purpose to criticise their reduction algorithm, but only to show how the SIMD style can be adapted readily to the simulation of a mixed SIMD-MIMD architecture

such as that of the SUPER machine. However, we would like to point out that their program does gloss over a number of issues which we have discussed in detail, especially as far as the reclamation of dead nodules is concerned. It is not necessary to interrupt the reduction processing to do garbage collection except when the reference-count based continuous reclamation falls behind the demand for new nodes.

Nor is it a trivial matter to determine efficiently whether the graph is in normal form. As our discussion of the matter immediately below suggests, we consider that the computational issue is to be able to turn the normal form property into a local property of the root of the graph rather than to let it remain (as it naturally is) a global property of the whole graph.

Finally, we of course would argue that it is necessary to introduce pointer, constant and quantification nodes as well as combination nodes, in order to have an efficient representation of the graph.

Their algorithm is then the following:

```
while [graph not yet in normal form] do
|   for each combination node n in parallel do
|   |   opt1 := OPERATOR [n]
|   |   if COMBINATION [opt1] then
|   |   |   opd1  := OPERAND [n]
|   |   |   if  COMBINATION(opd1) and  OPERATOR [opd1] = I  then
|   |   |   |   OPERAND [n ] := OPERAND[opd1]
|   |   |   fi
|   |   |   opt2 := OPERATOR[opt1]
|   |   |   if  opt2 = K  then  OPERATOR [n] := I ,  OPERAND[n] := OPERAND[opt1]  fi
|   |   |   if  opt2 = I  then  OPERATOR [n] :=  OPERAND [opt1]  fi
|   |   |   if  COMBINATION (opt2)  and  OPERATOR[opt2] = S   then
|   |   |   |   a := new ; b :=  new
|   |   |   |   OPERATOR [a]  :=   OPERAND [opt2] ;  OPERAND [a] := opd1;
|   |   |   |   OPERATOR [b]  :=  OPERAND [opt1] ;  OPERAND[b] :=  opd1 ;
|   |   |   |   OPERATOR [n]  :=  a ;  OPERAND [n] :=  b
|   |   |   fi
|   |   fi
|   rof
|   [ perform garbage collection if necessary]
elihw
```

and we believe that its concision and clarity are exemplary.

SUPER front-end monitors nodules to detect normal form. The front-end machine has itself an important role to play in the reduction process, namely, to watch its development and determine when it has reached a suitable stopping point. In the LNF-Plus system, the reduction continues until the lazy normal form is attained by the expression graph (see **Volume 2, 3.4**). It will be recalled that lazy normal form consists of not containing an initial redex; other (non-initial) redexes may well be present. This means that the "spine" of the graph is in its final form, but the "arguments" need not be. This condition can be detected by examination of the graph's spine alone. The spine of a meaningful graph is the longest sequence of vertices, starting at its root, such that each vertex is the operator of its predecessor. It follows immediately from this definition

that the spine is finite, and that its last element is an atom. This atom is called the initial atom of the graph. If the spine contains no redex (and it can contain at most one) then (by definition) the graph is in lazy normal form. So the front-end machine has to watch for the spine's becoming "empty" of redexes. The following discussion is a summary of that in **Volume 2, 1.3.3**.

The LNF field. The field LNF in the local memory of nodule i is set to **true** when the vertex i detects that i is in lazy normal form. That is to say, i observes that the following condition is true:

either	i is an atom
or	i is a pointer and the field LNF [WHERE [i]]is **true**
or	i is a non-redex combination and LNF [OPERATOR [i]] is **true**.

Each nodule i, as part of its routine behavior, monitors this condition and sets LNF[i] to **true** as soon as the condition becomes true.

The SUPER front-end machine need only watch the root's LNF field. Given the correct implementation of the above piece of nodule behavior, it is easy to see that the front-end machine need only watch the LNF field of the root nodule to be able to detect when the graph has reached lazy normal form. Compared with the lazy normal form reached by the single-reduction system LNF-Plus, the SUPER lazy normal form will be somewhat more "refined" in the sense that the arguments of the expression will have experienced many more reduction steps.

SUPER parallelism invisible to user. We emphasize again the point that data parallel computation has the very desirable feature that it feels to the user, sitting at the front-end machine, just like ordinary single-processor computation, the concurrency behind the scenes being invisible. In SUPER computations the user is aware only of the fact that the expressions given to the front-end machine as input evoke some kind of internal reduction process which results eventually in a suitable output expression. That this involves the cooordinated behavior of a very large number of processors is evident only indirectly from the speed with which the reduction takes place and from the fact that the arguments (in a lazy mode of reduction) are reduced somewhat further than would have been the case in a single-reduction machine.

This implicit parallelism avoids the extremely difficult problems of intellectual control over the complexity of a system of cooperating concurrent processes. To try to remain cognizant of the separate behavior of each processor in a large multiprocessor system is a severe strain on the limited human capacity to handle dynamically changing information patterns.

CHAPTER 4. RELATED WORK.

We are aware of other work on parallel combinator reduction by Simon Peyton Jones [5], Paul Hudak [18], and by Joseph Goguen and Jose Meseguer [11].

The ALICE project [9] at Imperial College, London, is a multiple-reduction parallel architecture for the λ-calculus. Mago [22] has a project to design a fine-grained parallel multiple-reduction machine for the λ-calculus. There are several projects to develop dataflow architectures which must be considered multiple-reduction machines for simple applicative systems which fall short of the full expressive power of the languages considered in this report. Only one of these, at the University of Manchester in England, has actually been built [14].

The earliest reduction machine architecture of any kind known to us is Klaus Berkling's λ-calculus single-reduction processor, the GMD Machine [3]. More recently there have been two **SKI**-graph single-reduction machines built by the SKIM group led by Arthur Norman in Cambridge, England [8], and another one built by the NORMA group at the former Austin Research Center of the Burroughs-SDC Corporation (now UNISYS) [25]. There is another project currently under way at the UNISYS Paoli Research Center to build a system similar to SUPER. The G-machine system [1, 19] at the University of Goteborg in Sweden involves compiling runtime code for a conventional von Neumann processor too carry out the reduction of a SKI-graph-like transform of the source expression.

Of course it is the pioneering, elegant systems of Turner [28] which have inspired much of the above work, and which certainly have been a major influence on our own.

REFERENCES.

[1] Augustsson, L. — A compiler for Lazy ML. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, 1984.

[2] Backus, J. — Can programming be liberated from the von Neumann style? The Turing Award Lecture for 1977, in Communications of the A.C.M., August 1978.

[3] Berkling, K. J. — Reduction Languages for Reduction Machines. Second International Symposium on Computer Architecture, 1975.

[4] Burge, W. H. — Recursive programming techniques. Addison-Wesley, 1975

[5] Clack, C. & Peyton Jones, S. — The Four-stroke reduction engine. Conference Record of the 1986 Symposium on Lisp and Functional Programming, 1986.

[6] Clark, K. L. — Negation as failure. In Logic and Databases, edited by Gallaire and Minker. Plenum Press, 1978.

[7] Clark, K. L. — Predicate logic as a programming formalism. Ph.D. Thesis, Imperial College, London, 1979.

[8] Clarke, T. J. W. et al. — SKIM - The S, K, I Reduction Machine. Conference Record of the 1980 Symposium on Lisp and Functional Programming.

[9] Darlington, J. & Reeve, M. — ALICE - a multiprocessor reduction machine for the parallel evaluationof applicative languages. Symposium on functional languages and their implications for computer architecture. Goteborg, Sweden, 1981.

[10] Fuchi, K. — Revisiting original philosophy of Fifth Generation Project Proceedings of the International Conference on Fifth Generation Computer Systems 1984. ICOT, Tokyo, 1984.

[11] Goguen, J. & Meseguer, J. Models of computation for the Rewrite Rule Machine. SRI Technical Report, July 1986.

[12] Greene, K.J. A fully lazy, higher order, purely functional reduction language with reduction semantics. CASE Center Report 8503, Syracuse,1985. (also **Volume 2** of this report).

[13] Greene, K.J. User's Guide to the LNF-Plus System. Syracuse University, 1987. (also **Volume 3** of this report)

[14] Gurd, J.R. et al. The Manchester prototype dataflow computer. University of Manchester Technical Report, 1979.

[15] Henderson, P. Functional Programming. Academic Press, 1979.

[16] Hillis, W. D. The Connection Machine. MIT Press, 1985.

[17] Hillis, W. D. & Steele, G. L., Jr. Data parallel algorithms. Communications of the A.C.M 29, 1986, 1170 - 1183.

[18] Hudak, P. & Goldberg, B. Distributed execution of functional programs using serial combinators. IEEE Transactions on Computers, Vol. C-34, 1985.

[19] Johnsson, T. The G-Machine: an abstract machine for graph reduction. Goteborg, 1983.

[20] Kowalski, R. A. Predicate logic as programming language. Proceedings of IFIP Congress 74.

[21] Landin, P. The next 700 programming languages. Communications of the. A.C.M. 9, 1966.

[22] Mago, G. A cellular architecture for functional programming. Proceedings of IEEE Computer Conference 1986.

[23] Robinson, J. A. A machine-oriented logic based on the resolution principle. J.A.C.M.,12, 1965, 23 - 41.

[24] Robinson, J. A. & Sibert, E. E. LOGLISP - an alternative to PROLOG. Machine Intelligence 10, 1982.

[25] Scheevel, M. NORMA: a graph reduction processor. Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming.

[26] Steele, G.L. Jr. & Sussman, G.J. SCHEME: an interpreter for extended lambda calculus. AI Memo 349, MIT, 1975.

[27] Stoye, W. R., et al. Some practical methods for rapid combinator reduction. Conference Record of the 1984 Symposium on Lisp and Functional Programming, 1984.

[28] Turner, D. A. A new implementation technique for applicative languages. Software Practice and Experience, 9, 1979.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END

DATE

FILMED

DTIC

4/88